Universitetet i Oslo
Institutt for informatikk

Scripting in High-Performance Computing

Roger Hansen

Cand Scient Thesis

August, 2001

Abstract

This thesis describes how the use of more than one type of programming languages can be useful in high-performance computing. Scripting languages like Python and Perl are used for control and computational steering, while compiled languages as C and Fortran are used for the numerically intensive tasks. How to efficiently call compiled code from scripting languages is discussed in detail, and numerous examples show how this can be done. The examples show that a combination of e.g. Python and Fortran is an option very much worth considering for numerical applications. The combination can result in applications with both run-time efficiency and flexible computational steering options. There are even good chances that development time for such applications compare very favorably with more traditional numerical applications.

Preface

This is the written part of my Cand. Scient grade at the Department of Informatics, University of Oslo.

I would like to thank some of my friends who helped and supported me during the last years. Many thanks to my friends and colleagues at MiP for moral support and constructive criticism. Especially Andreas Nergaard and Sigurd Larsen has been very helpful reading early manuscripts, and Igor Rafienko for always answering my strange questions about C and C++ programming. Many thanks to Thomas Skjønhuag, Kent-André Mardal and Ola Skavhaug for reading early manuscripts and giving constructive criticism and lots of interesting discussions. Many thanks to other friends, fellow students and my family for support and making studies and life enjoyable. Special thanks to Elise Thorbjørnsen for her love and support the last two years.

Many thanks to all persons who are involved in the Linux, GNU and Open Source communities, in particular Richard Stallman and Linus Torvalds. Thanks for your idealism and using your valuable time in making superb programs free for others to use.

Finally, I would like to thank Hans Petter Langtangen for being a great boss, mentor and friend. Without his inspiration, patience and guidance I could not have finished this project.

Roger Hansen, Oslo, July 2001

Contents

Chapter 1

Introduction

In this thesis we will study how the fields of numerics and high-performance computing can benefit from the use of more than one type of programming language in a single application. Due to their high run-time efficiency, primitive languages like Fortran and C have been dominating high-performance computing, and are still in widespread use. Modern scripting languages have until recently been little used in high-performance computing, simply because numerically intensive programs written in them run too slowly to be practical. We will show how modern scripting languages can enhance numeric software by easing the creation of flexible *problem solving environments* in which the scripting languages are used to interface C or Fortran code. We will try to show how one can combine the best features from both worlds, and thereby create programs that satisfy the need for efficient use of the available hardware and development time through the use of a flexible problem solving environment.

Scripting languages are intended to be interactive, dynamic, portable and to offer high-level data types and structures. They are ideal for creating small and useful applications, but some of them are also suitable for creating large systems. Compared to equivalent systems written in primitive languages like C, the development time for such systems is usually far shorter, see e.g. [19] or [21]. A dynamic, interpreted language cannot produce as fast machine code as a compiler for a primitive, static language, since there is less information available at code translation time. This is why scripting languages can be slow for numeric calculations, as shown in the introductory example in Chapter 1.1 where we compare the speed of array operations for a few scripting languages and some compiled languages.

Primitive languages like Fortran and C offer tiny abstractions over assembly code, and the compilers produce very fast code. Especially Fortran has been, and still is popular in communities of scientific and high-performance computing. In the recent years object-oriented languages like C++, Java and Fortran 95 have been used to some extent, giving programmers better tools for creating useful abstractions. With these languages programmers can create more flexible software, but they still lack the dynamic and interactive nature of the scripting languages.

So, in this thesis we will show that it is possible to use the best from both worlds. We will create and discuss applications which use scripting languages for design and control, and use primitive compiled languages for speed demanding tasks. Through our examples we will show that freely available tools can greatly simplify the process of mixing languages in an application. In fact, a combination of e.g. Fortran and Python code can be very efficient considering both CPU run time and development time. We

will show two strategies for combining two or more languages in numerical applications:

1. *Mixed-language numerical code.* That is, we write new code mainly in a scripting language, but perform numerically intensive calculations in a fast compiled language.

2. *Access to existing numerical code.* That is, we want to call existing numerical libraries or applications directly from a scripting language.

These strategies are useful in a much wider context than just numerical software. Any problem that contain tasks best solved in different languages may benefit from one of the strategies above. In both strategies we will need communication between code written in different languages. Such situations almost always imply the need for wrapper code, i.e. code that handles language differences so that control and data can pass between sections of code written in different languages. Python will be used as our main scripting language in this thesis, but others can be used for the same purposes. Perl and Tcl are discussed briefly below, and some other possible alternatives are mentioned in Chapter 4.1.

In the next section we investigate the efficiency of array operations for both compiled and interpreted languages. Chapter 2 is an introduction to *Mixed-Language* programming. This chapter includes a discussion of code wrapping concepts, and contains a non-trivial code example that illustrates some code wrapping techniques. In Chapter 3 we discuss *problem solving environments* and *computational steering*, where we use a wave simulator as our main example. Experiences from wrapping large numerical libraries are discussed in Chapter 3.4. We end this thesis with some concluding remarks and suggestions for further work with these topics in Chapter 4.

Please note that several parts of this thesis have been developed in cooperation between the author and Hans Petter Langtangen. The text in chapters 2.3.1 and 3.1.1 are mostly written by Langtangen. The text and code in Chapters 1.1, 2.3.2–2.3.6, and 2.4.1–2.4.2 are joint work. These chapters will appear in the book [17]. Notice that the source code described is an important part of this thesis. Much time and effort has been spent on making the code useful, efficient, and as readable as possible, but understanding all of the code is not neccessary for the overview. Appendix A lists the software needed for running the code, and also contains a listing of the source files developed and discussed.

1.1 Timing Array Operations

Let us evaluate the efficiency of array manipulations in scripting languages versus typical compiled languages. The idea is to show how bad the scripting languages Perl, Python, and Tcl suffers when doing such calculations in an explicit loop, compared to the compiled languages C and Fortran. It is possible to do calculations more efficient in the scripting languages by using more specialized data structures, e.g. a numeric array instead of an all-purpose list, and by avoiding the explicit loop. This is shown for Python using the Numeric module.

Our simple test problem consist of evaluating a mathematical function at discrete points in the interval $[0, 1]$ and storing the function values in an array. The function to be used in the example reflects the velocity of a pressure-driven power-law fluid between two plane walls:

$$f(x) = \frac{n}{1+n} \begin{cases} 0.5^{1+1/n} - (0.5-x)^{1+1/n}, & 0 \leq x \leq 0.5 \\ 0.5^{1+1/n} - (x-0.5)^{1+1/n}, & 0.5 < x \leq 1 \end{cases} \tag{1.1}$$

We remark that n is a real number, typically $n \in (0, 1]$.

1.1.1 Timing Results in Python

The function `f` for evaluating (1.1) has been implemented in a Python script `fillarr.py`, found in the directory `src/ex/fillarr/script`. There are two alternative functions in this script for creating an array of function values:

1. `fillarr_append`, where we append new function values to a list,
2. `fillarr_repeat`, where we create a list of fixed size and then use subscription to insert function values.

These two functions have the following forms:

```
def fillarr_append(n):
    list = []
    for i in range(n):
        x = float(i)/float(n-1)
        list.append(f(x))

def fillarr_repeat(n):
    list = [0.0]*n
    for i in range(n):
        x = float(i)/float(n-1)
        list[i] = f(x)
```

Running the functions 20 times with arrays of length 100000 shows that the list append approach is about a factor of 1.25 slower than working with the `[0.0]*n` list.

An implementation of `f` can look like

```
def f(x, n=0.3):
    c = 1.0 + 1/n
    cH = 0.5**c
    A = n/(1.0 + n)
    if x <= 0.5:
        c1 = (0.5 - x)**c
    else:
        c1 = (x - 0.5)**c
    return A*(cH-c1)
```

Note that setting `n=0.3` as a *default argument* is both convenient and effective in Python. If we want to use another value for `f`, just send it as a second argument, e.g `f(x, n=0.4)`. The built-in power operator is used instead of the built-in function `pow` or `math.pow` since it is more efficient in this setting.

Using this optimal `f` function, the output from the `fillarr.py` script, run with Python 2.0, became[1]

```
fillarr_append: elapsed=49.5316, CPU=49.23
fillarr_repeat: elapsed=40.32, CPU=39.94
```

[1] My laptop was used for the tests; IBM 570E, Intel 500 MHz processor with 128 Mb RAM, running Debian Linux.

Function calls are rather expensive in Python, which is measurable when a function is called in a loop. If we in this example replace the function call in the for-loop, and simply do the calculations inline, we get a better result

```
fillarr_repeat_no_f: elapsed=32.50, CPU=32.15
```

The best Python approach is about 1.4 slower than the similar Perl version, see Chapter 1.1.2. When we eliminate the calls to `f` and do the calculations inline the speed of Perl is almost reached, but the code is less readable. Using Python's module for numerical computing (`Numeric`), we can easily reach the speed of C.

Using NumPy Arrays for Speeding up Python Code

The efficiency of the array operations in `fillarr.py` can be dramatically improved by replacing the loops by their equivalent NumPy expressions operating on arrays from the module `Numeric`. Basically, our script `fillarr.py` in standard Python calls a user-defined function for each entry in an array. Such a code must be replaced by a set of operations on the whole array at once. For a simple function like

```
def myfunc(x):  return 3*x + sin(2*x)
```

one can just call the function with `x` as a NumPy array[2]. However, our function (1.1) is a bit more complicated. We also want to do operations in-place to minimize memory requirements. A possible solution is listed next.

```
from Numeric import arrayrange, multiply, subtract, power, add, array
def fillarr_Numeric(size, n=0.3):
    x = arrayrange(size)*1.0/float(size-1)
    # now x contains the x values,
    # manipulate so x contains f(x) values:
    subtract(x, 0.5, x)
    multiply(x[0:size/2], -1.0, x[0:size/2])
    c = 1 + 1.0/n
    power(x,c,x)
    multiply(x, -1, x)
    cH = 0.5**c
    A = n/(1.0 + n)
    add(x, cH, x)
    multiply(x, A, x)
```

This function is implemented in the script `src/ex/fillarr/NumPy/fillarr1_NumPy.py`. Timings show that the CPU time on my laptop is reduced to 4.8 seconds, which is almost 10 times faster than using a loop over a Python array and quite close to the timings obtained in C, C++, or Fortran 77 code.

There are other alternatives as well in many scripting languages. Python offers some functional programming tools like `map` and `lambda` which can be used instead of a loop or NumPy operations. The code

```
def fillarr_map(size):
    x = arrayrange(size)*1.0/float(size-1)
    res = map(f, x)
```

[2] We remark that this will create temporary arrays holding intermediate results, here `2*x`, `sin(2*x)`, and `3*x`.

does exactly the same as the examples above. The built-in function `map` takes two arguments, a function and a sequence (e.g. list or NumPy array), and returns the result in a list. The sequence should contain the functions arguments. This is often much faster than using a loop, as in e.g `fillarr_append`. This function is about twice as fast as the functions with explicit loops, but still seven times slower than the Numeric version above.

Later, in Chapter 2.4, we will see that it is easy to write an efficient standard loop over the NumPy array entries in C, or Fortran as an alternative to using the vector-oriented `Numeric` functions only as we do in the presented `fillarr1_Numeric` routine.

1.1.2 Timing Results in Other Scripting Languages

Timing Results in Perl. The Perl code following have the similar functionality as in the corresponding Python scripts in Chapter 1.1.1 and shown here for complete reference. Notice that different improvements are available in Perl too, but for simplicity we only implement the loop functions.[3]

```
sub fillarr_append {
    my $n = $_[0];  # size of list
    @list = ();
    my $i;
    for ($i = 0; $i < $n; $i++) {
        $x = $i/($n-1);
        push @list, f($x);
    }
}

sub fillarr_repeat {
    my $n = $_[0];      # size of list
    @list = (0) x $n;   # create $n entries
    my $i;
    for ($i = 0; $i < $n; $i++) {
        $x = $i/($n-1);
        $list[0] = f($x);
    }
}
```

We run these subroutines 20 times with arrays of length 100000 as in the section above. The output on my laptop, running Perl 5.6.0, was

```
fillarr_append: elapsed=31, CPU=30.7
fillarr_repeat: elapsed=29, CPU=29.16
```

The conclusion is that the two ways of working with arrays are not significantly different with respect to computational efficiency.

Timing Results in Tcl. Inserting function values from (1.1) in Tcl can only be done by appending the values to a list:

```
proc fillarr_append { n } {
    for {set i 1} {$i < $n} { incr i} {
        set x [ expr 1.0*$i/($n - 1) ]
        append list [ f $x ]
    }
}
```

[3] Notice that division of two integers, like `$i/($n-1)`, results in the correct real number in Perl. This is not the case in Python, C, C++ and many other languages.

```
# run fillarr_append 20 times with list of length 100000:
set nr 20;  set length 100000;
set txt [ time {fillarr_append $length} $nr ]
```

Running this example on my laptop gave about 800 seconds CPU time (Tcl v8.0), which is dramatically slower than the Perl and Python versions. These tests shows that Perl are a little faster than Python when doing explicit loops, and that they are much faster than Tcl for our problem. However with Python's Numeric module, or by avoiding the explicit loop the problem can be solved much faster. Next we will measure the efficiency of C, C++ and Fortran.

1.1.3 Timing Results in C and Fortran 77

It is of great interest to see the CPU times of a corresponding C or Fortran 77 program that fills an array of length 100000 with function values from (1.1) 20 times. A possible C implementation is listed at page 38 in Chapter 2.4.1. This program is essentially doing the same as we did in the Perl, Python, and Tcl scripts. The C version used only about 3 seconds of CPU time on the same machine. This is 13 times faster than Python and 10 times faster than Perl, and about 1.6 times faster than the NumPy version. A similar implementation in Fortran is listed at page 39 in Chapter 2.4.2.

On the Linux machine where the tests were performed, Fortran 77 ran at approximately the same speed as C (although the Fortran version avoids dynamic allocation of 20 arrays of length 100000 – this task took very little time). On a Sun machine running Sun's native Fortran 77 compiler, the Fortran version was almost a factor of two faster than the C version. Note that on Sun, the programs were compiled using the `-fast` option for optimization. Native `f77` and `CC` (version 4) compilers were used.

It is interesting that for a simple problem like this, the compiled languages are so much faster than the scripting languages when the problem apparently is solved in an equivalent manner. But there are important differences to be aware of, even if the code seems similar. The list data type for the scripting languages are "all-purpose" dynamic lists with lots of features. This is very different than the simple contiguous memory blocks that C and Fortran arrays really are. Another important issue is that the C and Fortran compilers can optimize the loop and computations. The Python, Perl, and Tcl interpreters can not optimize as much, since there is less information available at code translation time.

Later in Chapter 2.4 we will discuss implementations evaluating (1.1) that combine Python code with C and Fortran. Note that we are not running a complete benchmark of the efficiency of array operations, but only giving an impression of the relative speed between a few languages for a typical numerical problem.

Chapter 2

Introduction to Mixed-Language Programming

There are a lot of programming languages designed for different purposes, all with their strengths and weaknesses. No language is suited for all purposes, and thus using code written in different languages in an application can be useful. Using this mixed-language strategy almost always imply the need for *wrapper code* customized for the languages involved. The wrapper code must take care of necessary type conversions and possibly other tasks.

In the next section we will discuss the concepts of code wrapping, with emphasis on the programming languages used in this thesis. Automatic generation of wrapper code is discussed in Chapter 2.2, and a short introduction to some specific tools are presented. Chapter 2.3 contains a computational intensive, but quite simple example where we discuss pure Python and C/C++ implementations versus implementations combining Python code with code written in C, C++, and Fortran. This example serves as an practical introduction to code wrapping. In Chapter 2.4 we apply code wrapping to the examples presented in the introduction, and compare the efficiency of the different implementations. Some alternative methods for inter-language communication are briefly discussed in chapter 2.5.

2.1 Code Wrapping Concepts

Wrapper code is often necessary to enable communication between code written in different programming languages. Such code should handle the differences between the languages, e.g. conversion between data types. What the wrapper code should look like depends on the programming languages, and sometimes the operating system (OS) where the code is running. In general we could say that it is easier to mix languages that have much in common, but some languages have methods for translating code to or from another language. The programming languages used in this thesis are described in the next section, followed by a section describing the code wrapping process. We will refer to the language of the code to be wrapped as the *target language* (TL), and the language which use the wrapped code as the *application language* (AL). The application language is sometimes called the extended language, or the AL is said to be extended.

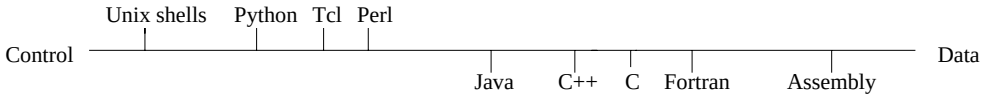

Figure 2.1: Programming languages: Control versus data. Unix shells are typical command languages, while Perl, Python, and Tcl are scripting languages. C, C++, Fortran, and Java are said to be structured languages. Command and scripting languages are typically more control oriented than structured languages.

2.1.1 Programming Languages

In this section we will study the languages used in this thesis. We will not give a thorough description of them, but describe the most important properties and features that concern our context. This includes a language's basic data types, other data types and structures, typing, and special constructs that makes wrapping easier or more difficult. Some basic concepts of programming languages are discussed. We will study *Fortran, C*, and *C++* as target languages, since they are the most used programming languages in high-performance computing. As application languages we will study Perl, Python, and Tcl, since they are some of the most popular scripting languages.

It is possible to turn this the other way around, using the target languages mentioned above as application languages and vice versa, but for high-performance computing the setting above makes most sense. The main reason is that we want to write most parts of the application in a high level language and the time critical parts in the fastest code possible. This is why all of our target languages are compiled languages with static type binding, which produces the fastest code possible[1], and our application languages are dynamically typed and interpreted languages. Figure 2.1 illustrates the relative level between some languages. High-level languages are control oriented, and low-level languages are more data oriented.

It is important to distinguish between a programming language and its implementations. A language implementation could be a compiler or an interpreter depending on the specific language. Most programming language standards do not directly describe how communication with other languages should be done. The implementations of a programming language, on the other hand, almost always have one or more recipes for such communication. Python, for instance has two widespread implementations with different systems for inter-language communication. The most widespread of them is the C implementation, sometimes referred to as CPython to specify that it is written in (ANSI) C. This implementation has an API which has utilities for conversion of data types, type checking and other useful services, and naturally communication with C is well supported. The other implementation, often called Jython (or JPython), is implemented in Java, and Java libraries can be imported straightforwardly. But communication with code written in other languages is treated differently. This is also the situation for several other languages with more than one implementation. Note that we sometimes mention a language implementation with only the name of the language, e.g we refer to "the C implementation of Python" as Python.

In the discussion of data types we must distinguish between basic or built-in types and advanced types. Basic data types as integers and characters exist in all the languages

[1]Except for highly tuned assembly code, but writing assembly code most often has a very poor time spent versus effect gained ratio. Besides compilers continually improve.

discussed here, but advanced or high-level data types such as lists or hash tables[2] do not. Conversion of primitive data types between two languages is usually easy, since the data type is simple and the implementation of them do not differ much. If two data types are implemented in an equivalent manner we say that they are *isomorphic* or that we have *isomorphism*. For more advanced data types conversion can be extremely difficult, especially if the data types do not match very well. In such cases we say that the data types are *non-isomorphic*.

Another important concept is how a language is typed, i.e whether it has *dynamic* or *static typing*. Static typing means that a variable must be declared as a certain type, which can not be changed. Dynamic typing means that the type of a variable can not be determined before run time, and that it may change during program execution. Fortran is an example of a language with static typing. If a variable in Fortran is declared as an integer it must remain an integer. Python on the other hand is dynamically typed, and a variable can be an integer at first and then become for instance a string or an object. For more information about programming languages, see e.g "Programming Language Concepts" [12].

Fortran

Fortran, developed between 1954 to 1957, was the first third generation programming language, and was designed for numerical computing. It was developed in a time with very primitive hardware by today standards. Computers were slow, and had very little memory. Thus Fortran is a primitive low-level language with static typing and memory allocation, which makes it efficient in the sense of speed and memory usage.

Fortran exists in many versions with some variations in syntax, because compilers where implemented differently. As a result, standards there has been developed standards like Fortran 77 and Fortran 90. In this thesis we will mostly discuss Fortran 77, since it is by far the most used of the Fortran standards.

Fortran has the primitive data types

- integer
- real
- double precision
- complex
- logical
- character

where real and double precision are floating point numbers. Fortran has an *array* data type that can consist of a contiguous linear sequence of elements of one of the basic data types. Fortran allocates a contiguous block of memory equal to the size of the basic data type multiplied by the number of elements in the array. No array bounds checking is offered. Fortran 77 allows arrays of up to seven dimensions, which are stored *by column* unlike arrays in most other languages. This complicates the wrapping process of Fortran arrays to or from languages with row-wise array storage.

[2]Hash tables are also referred to as associative arrays and dictionaries.

Fortran is by today's standards a small and very simple language with few data types and language constructs. Fortran has no global variables, i.e. variables that are shared among several program units (subroutines). A way of sharing variables between different subroutines is the use of common blocks. This construct and column-wise array storage can make wrapping a bit difficult, but still Fortran is nearly ideal as a target language for wrapping purposes, in a numerical context.

C

The C programming Language, created in 1972, was designed for system programming purposes. It is compiled, static typed and low-level like Fortran, but offers some more data types and constructs. Basic types like `char`, `int`, `float`, and `double` are offered. In addition, some qualifiers can be applied to the basic types. For instance `short` and `long` apply to integers. The intent is that `short` and `long` should provide integers with different length, which depends on the hardware and OS. The idea is that other data types can be expressed through the basic types.

The concept of pointers makes C different from Fortran, which offers no such construct. Pointers in C are extremely flexible since they can point to anything in memory. Pointers can be very useful, but can be a major source of obscure errors. Sometimes pointers complicates the wrapping process. With a function prototype like

```
void foo(double a, double b, double *c);
```

it is difficult to know what purpose the pointer actually has. With another prototype like

```
void add(double a, double b, double *result);
```

it is reasonable to expect that `result` points to where the function is expected to place the result of `a+b`.

Arrays in C have much in common with Fortran arrays, except that they are stored *row-wise*. In addition there is a close connection between pointers an arrays in C, meaning that every expression including arrays can be written with pointers instead. Another useful C construct is the *struct*. Structs group variables into a collection, and can contain variables of any data type available, including pointers and other structs. Wrapping structs is usually not a problem since they most often can be mapped to objects in an object-oriented programming language.

Even though C has pointers, structs, and support recursion it is considered a low-level programming language. Optimized C code is almost as fast as Fortran code, and probably is the second most used language in scientific and high-performance computing. In most cases C is a good choice as a target language, since it is relative easy to wrap and since many wrapper code generating systems support it. The language is described in detail in the classic book of Kernighan and Ritchie [15].

C++

The C++ programming language, created by Bjarne Stroustrup in 1984, was developed as an extended C, supporting the object-orientation (OO) concept. Even though C++ is based on C it is a very different language. First of all it is a much larger and more complex language, with many more data types and constructs. Another interesting

fact is that it supports both the procedural, object-oriented and generic programming paradigms, and thus can be said to be a high-level language. Fortran and C are mainly procedural languages.

Since C++ has static typing and no run-time system, C++ code can be very efficient if the code and compiler are good. Unfortunately the vast complexity of the language makes it extremely difficult to develop compilers that can handle all C++ features satisfyingly. In fact most C++ compilers are not able to process all the advanced template expressions allowed by the language standard. Thus most programmers use only a subset of C++. On the other hand, it is possible to use only a subset of C++ and still create useful applications, and C++ has become quite popular in scientific and high-performance computing communities the last few years. This is mainly because C++ offers lots of useful abstractions and that the code can be very efficient. Because of its complexity, C++ can be extremely difficult to wrap, but the simpler constructs are possible to wrap, and thus C++ can be useful as a target language in some cases. See "The C++ Programming Language" [23] for a complete description of C++.

Python

Python is a high-level dynamically typed and interpreted language, created about ten years ago by Guido van Rossum. It is still being developed by the PythonLabs team lead by van Rossum. Python is a scripting language, but has turned out to be useful in a much wider range of applications. In this thesis we will use Python for numeric computations with fairly good results, see e.g. Chapter 2.3 and Chapter 3.

Some of Python's strengths are the simplicity, readability, and ease of use. Python is not particularly fast, not even for a interpreted language. The Perl interpreter, for instance, is faster than any python implementation for many types of operations. One of the main advantages of Python is that it supports communication with other languages very well, as it was designed also to be easily *extensible*, i.e being capable of using code written in other languages.

Python has the same basic data types as C, and several more. Types as long (arbitrary precision integer) and complex numbers, strings, lists, tuples, and dictionary are all built-in data types in Python. All types in python are objects, and can be separated into *mutable* and *immutable* objects. If an object's value can be modified, it is said to be mutable. Lists in Python are mutable, but number types are not. If a number is assigned to a variable `i`, the variable gets a reference to an number object with correct type and value. If then a new number is assigned to `i`, the first reference is lost and `i` gets a reference to a new number object.

Python's concept of basic types is different than C's, we say they are non-isomorphic. But Python's basic types are based on the basic types of C, and Python's C API defines methods for conversions. Furthermore, the basic types in C are defined equivalently with the basic types of many languages, say C++ and Fortran. Thus we can use or rewrite some of the methods for communication to those other languages. These facts makes Python well suited as an application language in our context. Documentation of Python can be found at the Python home page [8] or Beazley's reference book [5].

Other Scripting Languages

There are other scripting languages like *Perl* and *Tcl* which in many ways are similar to Python. They are interpreted and dynamically typed languages with many of the

same high-level data types and structures. As Python, they are both extensible. There are differences both in syntax, philosophy, and intended use, but they still belong to the same category of programming languages. Thus Perl and Tcl could also be good choices as application languages, but Tcl has shown to be slow for numerical calculations, as we saw in Chapter 1.1.2. So, if Tcl should be used in our context, efficient modules must take care of the numerics. Perl, on the other hand, is often faster than Python, but we will use Python as our main application language for the following reasons:

- Python has an active community doing scientific and high-performance computing.
- Python has the Numeric module with NumPy arrays, and other useful modules for numeric computing as well.
- Python has a simpler and clearer syntax than Perl and Tcl, and is by many programmers considered to be better suited for large scale applications.

In the conclusion at page 81 some other interesting languages are mentioned as possible alternatives to Python. `Common Lisp` and `Ruby` are perhaps the most interesting languages for our purposes. In the rest of this thesis we will use Python as the application language, but we remark that other languages can be used with satisfying results.

2.1.2 The Wrapping Process

The process of wrapping code in a target language, and using this code in an application language relies on three parts. We have

1. the code to be wrapped, written in TL.
2. the wrapper code.
3. the application code, written in AL.

Sometimes the code to be wrapped is not developed with the application code. It can be a library written by a third party, and maybe just available as compiled code. In such cases it is not desirable or possible to change this code. For the application code we may have a similar situation, i.e it may be better to rewrite as little code as possible. As a general rule the wrapper code should take care of all the dirty details, and adapt to the other parts. If all the code is written more or less from scratch other adjustments may be preferable.

There are several strategies for writing wrapper code, depending on the application language and the implementation of that programming language. We can divide into three main strategies for writing the wrapper code.

- Write the wrapper code in the target language.
- Write the wrapper code in the application language.
- Write the wrapper code in some other language

Which strategy we choose depend on languages, preferences, and the tools available. It is possible to combine the strategies mentioned above. We will discuss the consequences of and the use of these strategies in the next sections.

Write the Wrapper Code in the Target Language

Writing the wrapper code in the target language is sometimes a good strategy. This solution is most useful when the target language is identical to the operative system language (i.e. the language used to implement the OS), or communicate straightforwardly with the OS. Then the wrapper code can be compiled and used in the same manner as other OS services. This solution is best suited for low level services, and not for the use we are mostly aiming at in this thesis.

Sometimes the target language is the same as the language used to implement the AT. This is the situation when we use C as TL and Python (i.e. CPython) as AL. In such cases this can be a very good solution, as we will see examples of in Chapter 2.3 and 3.1.

Write the Wrapper Code in the Application Language

Sometimes it is possible to write all the wrapper code in the application language. This is only possible if we have isomorphism between data types in AL and TL, or data type conversion can be done entirely from the application side. This is not very common but is supported by some languages. Perl with its Inline module is one example. The module let users write code in TL directly in the Perl code, and the module, which is mostly written in Perl creates the wrapper code. A small example from the documentation (see [14]) shows how a little C function can be used:

```
use Inline C => <<'END_C';
void greet() {
    printf("Hello, world!\n");
}
END_C

greet;
```

This script will, when run in a Perl interpreter, print: "Hello, world".

Write the Wrapper Code in Some Other Language

One of the most usual strategies is to write the wrapper code in the same language as the implementation of the application language. This is how we write wrapper code for CPython, and where the TL is not C (then we have the situation in Chapter 2.1.2). An example of this strategy used in this thesis is wrapping of Fortran code for use in a Python application, see Chapter 2.3.6. In that situation Fortran is TL, Python is AL, and the wrapper code is written in C.

2.2 Automatic Generation of Wrapper Code

As we have seen in the sections above, and will see in examples in the following chapters, wrapper code follows strict rules according to the languages in use and their implementations. A specific set of operations like type checking, data type conversion, and other tedious tasks must be done for each data type, and every other language construct used. This often results in lengthy monotonous code, where one of the main challenges is to avoid typographical errors.

If we have isomorphic data types or methods for converting non-isomorphic data types the wrapping process can be well enough defined to be done automatically. For

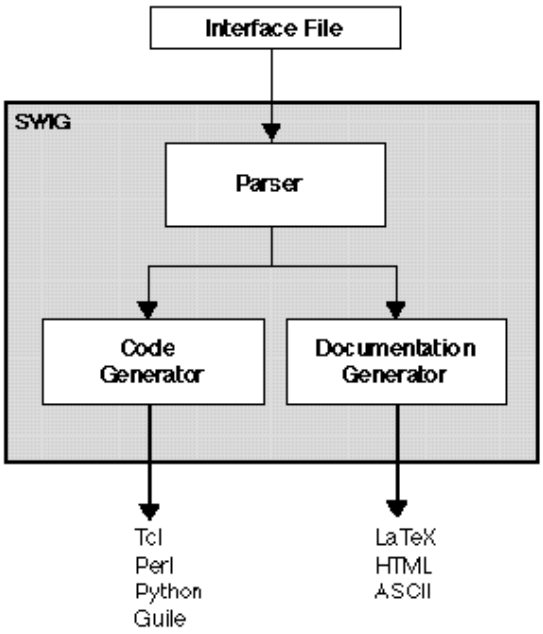

Figure 2.2: Illustration of the wrapping process with SWIG.

non-isomorphic data types and language specific constructs additional information must be given for the process to succeed. Fortunately there are several automatic wrapper code generating systems available. Most of them work by parsing the code to be wrapped, and reading a user-written interface description. Many also do conversion of non-isomorphic data types provided that a correct data type mapping is defined in the interface description. We will give a short introduction and description of some systems below. In later chapters we will use some of the systems presented.

2.2.1 Introduction to SWIG

SWIG is an acronym for *Simplified Wrapper Interface Generator* which is a tool used to make C and C++[3] code usable from scripting languages like Perl, Python, and Tcl. SWIG works by reading a interface file (which the user must write), parse this file and generate wrapper code and documentation, see Figure 2.2. Instead of writing wrapper code manually, one must write an interface file with a C like syntax. The latter is in most cases much simpler and less work.

Header and code files can be included or inserted directly into the interface file. If the code is simple enough, this is enough for creating the wrapper code. For more complicated constructs like pointers, structs, and conversion between non-isomorphic data types certain SWIG directives must be given in the interface file. SWIG has a mechanism called *typemap* for dealing with non-trivial data types and conversions between non-isomorphic data types. This does exactly what it says, either convert a data type from the target language to a new data type in the application language, or the other way around. As an example a list of structs in C code, implemented with pointers or an array, could be mapped into a list of objects in Python. We will see

[3]At least naive C++ code. Many C++ features, e.g templates, are not well handled by SWIG.

examples which use typemaps in Chapter 2.4.1. Typemaps can be used for:

- mapping of default value function arguments in AL to explicit arguments in TL.
- mapping of function return values from TL data type to AL data type
- mapping of a function argument from the AL data type to TL data type.

A small example inspired by the SWIG users manual (see [4]) will illustrate how SWIG works. Suppose we have some C code in the files: `example.h` and `example.c`

```
#include "example.h"

int fact(int n) {
  if (n <= 1) return 1;
  else return n*fact(n-1);
}

int mod(int n, int m) {
  return (n % m);
}
```

where the header file just contains function prototypes. The SWIG input file looks can like this:

```
%module example
%{
#include "example.h"
%}
%include "example.h"
```

The compilation process creates a shared object which can be imported as a module in python.

```
linux> swig -python example.i
Generating wrappers for Python
linux> gcc -c example.c
linux> gcc -c example_wrap.c -fpic -I/home/rogerha/ext/linux/include/python2.0
linux> gcc -shared example.o example_wrap.o -o examplemodule.so
```

Testing of the module can be done in the interactive Python interpreter.

```
Python 2.0 (#1, Jan 11 2001, 11:56:43)
[GCC 2.95.2 20000220 (Debian GNU/Linux)] on linux2
Type "copyright", "credits" or "license" for more information.
>>> import example
>>> example.fact(5)
120
>>> example.mod(23,7)
2
```

We will use SWIG in several examples for wrapping of C and C++ code in this thesis, see Chapter 2.3.5.

2.2.2 Introduction to FPIG

FPIG stands for Fortran to Python Interface Generator, and are sometimes called f2py. It makes wrapper code for Python, where the target languages are Fortran 77, Fortran 90 and Fortran 95. FPIG works pretty much like SWIG. It scans some given Fortran code and automatically writes the wrapper code. The nice thing with FPIG is that it even creates the interface file, which has borrowed its syntax from Fortran 95. Sometimes one has to modify that file, but for users familiar with Fortran 95 that will be fairly easy.

Another nice feature is that FPIG has an automatic connection with the NumPy arrays from the Numeric module. When a NumPy array is given as a parameter to a wrapped Fortran function, it will automatically work on the NumPy arrays data field in the Fortran function. With SWIG we must write a typemap to achieve this effect. We will use FPIG in Chapter 2.3.6. For more information about FPIG, see [20].

2.2.3 Other systems

A short introduction to some other wrapper code generators is given below. Some of these systems will be used later in this thesis.

Pyfort

Pyfort is another Fortran to Python interface generator. But this system concentrates only on Fortran 77. Pyfort works almost exactly like FPIG, but there is one major difference. Pyfort do not generate the interface file, so the user must write it manually, like with SWIG. The syntax of the interface file is almost equal to the syntax of FPIG interface files. We will use Pyfort in some examples in Chapter 2.3.6. For more information about Pyfort, see [10].

CXX

CXX is designed to make it easier to extend Python with C++ code, but has another philosophy than SWIG. Instead of creating the wrapping functions automatically, CXX makes the process of writing wrapper code easier. The most important difference between using CXX and writing the extensions using only Pythons C/API is that CXX offers a C++ interface to Python. So instead of doing all those tedious tasks in C, CXX offers an abstraction to this, and e.g hides the reference counting.

In short, CXX makes it easier (if you know C++) to write extensions from scratch. But still the wrapper code must be written manually, since this is not a parsing tool that generates wrapper functions. For more information refer to the CXX home page [11].

BPL

BPL (Boost Python Library) is a system for quickly and easily interfacing C++ code with Python such that the Python interface is very similar to the C++ interface. It is designed to be minimally intrusive on the C++ design. In most cases, one should not have to alter the C++ classes in any way in order to use them with BPL. The system should simply "reflect" the C++ classes and functions into Python.

Compared to the systems above BPL is somewhere between SWIG and CXX. It is used to interface C++ code, but the interface one has to write is an abstract layer over

Python's C API. Worth to notice is that the interface is written in C++, and thus can be linked directly with the library to e dynamic Python module. For more information, refer to the BPL homepage [1].

Siloon

Siloon is a very ambitious project that try to create a general system for parsing C, C++ and Fortran code, and produce interfaces to Perl and Python. It seems that the goal with Siloon is to create a complete Problem Solving Environment for the languages above. Unfortunately Siloon is not as mature as e.g SWIG and FPIG yet. For more information, see the Siloon homepage at `http://www.acl.lanl.gov/siloon/`.

Perl Inline module

Perl has a collection of Inline modules for automatic use of code in many target languages. The TL code can be placed directly in the Perl code (inline) and the module will silently create the necessary wrapper code. The Inline module supports C, C++ and Python, but the C++ support is rather poor. For more information, see e.g [14].

2.3 A Case Study for Mixed-Language Programming

This section is devoted to mixed-language programming, where Python scripts call up computationally intensive functions written in C, C++, or Fortran. Chapter 2.3.1 describes the computational problem. A pure Python implementation appears in Chapter 2.3.2. Extending of the Python scripts with graphics, run in a separate thread, is explained in Chapter 2.3.2. A pure C/C++ implementation of the Python script, without graphics, is presented in Chapter 2.3.3. The the efficiency of a Python script can often be significantly enhanced by utilizing NumPy functionality. This is exemplified in Chapter 2.3.4 for the current computational problem. We perform a profiling of the Python-based scripts to detect the computationally intensive parts and replace the code with C and C++ code in Chapter 2.3.5. In Chapter 2.3.6 we use Fortran code for the number crunching. These chapters shows how SWIG, FPIG and Pyfort can be used for code wrapping purposes.

2.3.1 The Computational Problem: Stochastic Simulation

Our main introductory example on integrating Python with C, C++, and Fortran code concerns a simple, yet realistic, application from structural reliability. The end deflection u of a cantilever beam, see Figure 2.3, can be expressed (according to simple beam theory) as

$$u = \frac{FL^3}{3EI}, \tag{2.1}$$

where L is the length of the beam, F is the end load, E is Young's modulus reflecting the elastic properties of the beam, and I is the moment of inertia reflecting the geometry of the cross section of the beam. In many practical circumstances neither E nor F is known exactly. If we know some statistics of E and F, we can compute the corresponding statistics of u and the probability that the deflection stays within some safe interval. For simplicity we shall assume that F, L, and I are known parameters, and that E is

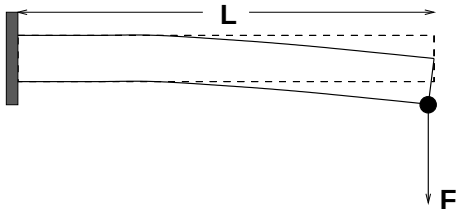

Figure 2.3: Deflection of a cantilever beam of length L subject to an end force F.

distributed as a Gaussian random variable with known mean and standard deviation. The purpose is to compute the mean, standard deviation, and probability distribution of u.

The technique for computing statistics of u employed here is Monte Carlo simulation. This means that we draw a large number of E values from a Gaussian distribution, compute the u value corresponding to each E value, and use simple statistical estimation techniques on all the u values for obtaining the mean, the standard deviation, and a histogram. The latter quantity acts as an approximation to the probability density of u.

Writing $u = f(E)$ with $f(E) = FL^3/(3EI)$ and letting $N(E_\mu, E_\sigma)$ be the normal distribution for E with mean E_μ and standard deviation E_σ, we generate random samples E_i, $i = 1, \ldots, n$, drawn from $N(E_\mu, E_\sigma)$, compute $u_i = f(E_i)$, $i = 1, \ldots, n$, and apply well-known formulas for the expectation u_E and the standard deviation u_S:

$$u_E = \frac{1}{n}\sum_{j=1}^{n} u_j, \quad u_S = \sqrt{\frac{1}{n-1}\left(\sum_{j=1}^{n} u_i^2 - n u_E^2\right)}. \tag{2.2}$$

2.3.2 A Pure Python Implementation

We shall first implement all parts of the stochastic simulation problem in pure Python code. We can thereafter migrate the most time critical operations to C, C++, or Fortran.

It is convenient to collect the basic statistical estimation techniques for the mean and the standard deviation in a Python class. As an option, the class can also store the samples in an array. This is convenient for histogram plotting or for computing other statistics than the mean and standard deviation as a post process. Notice that without the array of samples our simulation program can work with very large n values and avoid the computer's memory limitations.

```
class Statistics:
    def __init__(self, varname, n=0):
        self.sum1 = self.sum2 = 0.0 # sum of x, sum of x^2
        self.n = 0                  # current number of samples
        self.varname = varname  # name of the random variable
        if n > 0:               # store each sample?
            self.samples = [0.0]*n
            self.samples_length = n
        else:
            self.samples_length = 0

    def add(self, sample):
        "add a new sample"
        self.sum1 += sample
        self.sum2 += sample*sample
```

```
        if self.samples_length > 0:
            try:
                self.samples[self.n] = sample
            except IndexError:
                print "Statistics.add: no of samples is",self.n, \
                "which is larger than the declared size =", \
                self.samples_length
                self.samples.append(sample)
        self.n += 1

    def getMean(self):
        return self.sum1/float(self.n)

    def getStDev(self):
        return sqrt(1.0/float(self.n-1) * (self.sum2 -
                    self.sum1*self.sum1/float(self.n)))

    def getSamples(self):
        "return the samples collected so far"
        if self.samples_length > 0:
            return self.samples[:self.n]
        else:
            print
            "Statistics.getSamples: no samples are stored.\n" \
            "You need to call the Statistics constructor\n" \
            "with a specified (or maximum) number of samples!\n"
            return None

    def __str__(self):
        return "%6d samples: E[%s]=%g  StDev[%s]=%g" % \
               (self.n, self.varname, self.getMean(),
                self.varname, self.getStDev())
```

The `__str__` function defines the behavior of the `print a` statement when `a` is a `Statistics` object.

A typical application of the `Statistics` class is like this:

```
if n < 200:
    r_stat = Statistics('r', n)  # store samples
else:
    r_stat = Statistics('r', 0)  # no storage of samples
import random  # Python module for random numbers
# draw n Gaussian random numbers and compute their statistics:
for i in range(n):
    r_stat.add(random.gauss(2.0, 0.1))
print r_stat
if n < 200: print r_stat.getSamples()
```

The output is on the form

```
23000 samples: E[r]=1.99883  StDev[r]=0.100479
```

Having the deflection u implemented as Python function

```
def beam(F, L, E, I):  return F*L*L*L/(3*E*I)
```

we can easily write a Monte Carlo simulation as follows:

```
from Statistics import Statistics
from random import gauss

def MonteCarlo(n,         # no of simulations
               E_mean,    # mean of E
               E_stdev,   # stdev of E
```

```
            u_stat     # output Statistics object
           ):
    gauss_stat = Statistics("gauss")  # statistics of E
    for i in range(n):
        E = gauss(E_mean, E_stdev)
        gauss_stat.add(E)
        u = beam(1.0, 1.0, E, 1.0)
        u_stat.add(u)
        # write 10 intermediate results:
        if ((i+1) % (n/10)) == 0: print gauss_stat; print u_stat
    return u_stat
```

For simplicity we call `beam` with unit values for `F`, `L`, and `I`.

The usage of the `MonteCarlo` function can be like this:

```
import sys
from random import seed
from Statistics import Statistics

E_mean = 2.0; E_stdev = 0.2
try:    n = int(sys.argv[1])
except: n = 10000
# initialization of random generator by a 3-tuple:
seed((1,2,3))
u_stat  = Statistics("beam")
u_stat = MonteCarlo(n, E_mean, E_stdev, u_stat)
# n more simulations:
u_stat = MonteCarlo(n, E_mean, E_stdev, u_stat)
```

The complete code is found in `src/swig/sbeam/sbeam_py.py`. 200000 samples take about 13 seconds on my laptop, which is an acceptable speed in this example for practical purposes. Nevertheless, more demanding stochastic problems also demands more CPU power so it is interesting to see how fast NumPy-based and pure C/C++ codes are. Before we do that, we briefly show how easy it is to calculate and display a histogram of the u values in our Python script.

Adding Histogram and Graphics

A histogram representing the probability density of the beam deflection can be easily calculated using the Scientific Python module by Konrad Hinsen. To use the histogram functionality, we need to store all the samples so we send a second argument to the `Statistics` constructor:

```
u_stat = Statistics('u_stat', nsamples)
...
from Scientific.Statistics.Histogram import Histogram
h = Histogram(u_stat.getSamples(), 50)
h.normalize()  # let h be a density (unit area)
```

A plot of `h` would also be nice. Python's Gnuplot module by Michael Haggerty for the URL) allows easy access to Gnuplot from within a Python script. Suppose you have a list of data pairs with the points on some curve:

```
points = [[0,1.2], [1.1,5.2], [2,-0.3]]
```

The minimum requirements for making a plot are then

```
def plot(a, title=''):
    "a is a list of [x,y] pairs of data points"
    import Gnuplot
    g = Gnuplot.Gnuplot()
    d = Gnuplot.Data(a, with='lines', title=title)
    # let the Gnuplot object g plot the data object d:
    g.plot(d)
    g('pause 30')  # halt the plot for 30 seconds
```

The `Histogram` object has a data member `array` that holds a list of (x, y) points on the histogram curve such that plotting of the histogram in our example is easily enabled by

```
plot(h.array, title='Histogram of deflection')
```

at the end of the `MonteCarlo` function. These plotting statements are incorporated in the `sbeam_py.py` script that we have already referred to.

For our timing purposes it can therefore be convenient to turn on or off the histogram calculation and visualization. We have introduced an environment variable `SKIP_HISTOGRAM`, which equals 0 or 1, and controls whether a histogram should be computed or not:

```
if os.environ.has_key('SKIP_HISTOGRAM'):
    skip_histogram = int(os.environ['SKIP_HISTOGRAM']) # 0 or 1
else:
    skip_histogram = 0
if not skip_histogram:
    # histogram calculation and plotting
```

The effect of skipping the histogram calculation for 200000 samples was negligible on my laptop. However, the histogram calculation will consume a much larger portion of the CPU time when we use NumPy arrays in our simulation code.

Doing Graphics with Threads

The `plot` function has a major drawback: It halts the program flow for 30 seconds while displaying the graphics. The functionality we would like is to let the script proceed with the computations in the second call to `MonteCarlo` while the graphics of the first call is being displayed. In other words, we want the `plot` and `MonteCarlo` functions to be executed concurrently. This can easily be done using *threads*. Threads in Python behave much like threads in Java but are not yet as sophisticated.

Getting started with threads is easy. The program statements we want to run concurrently, or more precisely, in a thread separate from the main program, must be collected in a function, which is simply the `plot` function in the current example. The thread is constructed as (see [5, p. 174])

```
import threading
title = 'Histogram of deflection, n=%d' % u_stat.n
p = threading.Thread(target=plot, args=(h.array,title))
```

where `target=plot` indicates the function to be executed in the thread, and `args` is a tuple of arguments to be transferred to that function. Basic thread control includes starting, stopping, and restarting the thread:

```
p.start()
p.stop()
# do something with data
p.run()  # continue
p.stop()
p.run()
```

In the current example we only need to start the thread.

An alternative and more flexible implementation of a separate thread for the plotting statements makes use of a tailored class, e.g. called `PlotThread`, for running the thread. Class `PlotThread` must be derived from Python's `Thread` class in the `threading` module and provide a function `run` (without arguments) that performs the tasks in this thread. In the present example, `run` coincides with the `plot` function, except that the arguments to `plot` must be transferred in another way; we have transferred them through `PlotThread`'s constructor.

```
import threading
class PlotThread(threading.Thread):
    def __init__(self, a, title=''):
        "a is a list of [x,y] pairs of data points"
        threading.Thread.__init__(self)
        self.a = a; self.title = title
    def run(self):
        import Gnuplot
        g = Gnuplot.Gnuplot()
        d = Gnuplot.Data(self.a, with='lines', title=self.title)
        # let the Gnuplot object g plot the data object d:
        g.plot(d)
        g('pause 300')  # display the plot for 300 seconds
```

In the `MonteCarlo` we can then write

```
p = PlotThread(h.array, title=title)
p.start()  # start the thread, i.e., call p.run()
```

Both methods of implementing threads are included in the `sbeam_py.py` script.

2.3.3 A Pure C/C++ Implementation

The `sbeam_py.py` script can be implemented in C/C++ for increased efficiency. A practical drawback is that C and C++ does not provide convenient functions for drawing random numbers from the normal distribution. We therefore write a little module `draw` in C for doing this:

```
/* functions for drawing random numbers */

extern void setSeed(int seed);
/* draw random number in [0,1]: */
extern double draw01();
/* draw from the normal distribution: */
extern double gaussian(double mean, double stdev);
```

The corresponding files `draw.h` and `draw.c` are found in `src/swig/sbeam/c`.

The `Statistics` class from our Python script can be translated to C++:

```
class Statistics
{
  double  sum1, sum2;     // sum of samples and samples squared
  int     n;              // current no of samples
  const   char* varname;  // name of variable for statistics
  double* samples;        // optional storage of samples
  int     samples_length; // allocated length of samples array

 public:
  Statistics(const char* varname_, int n_ = 0);
 ~Statistics();
```

```
  void add(double sample);

  int getNoSamples() const { return n; }

  double getMean() const
    { return sum1/double(n); }

  double getStDev() const
    { return sqrt((sum2 - sum1*sum1/double(n))/double(n-1)); }

  const double* getSamples() const;

  void report(std::ostream& out);   // write statistics
};
```

This code segment is taken from the header file `Statistics.h`.

Central functions are `beam` and `MonteCarlo`, along with a kind of main program (the `run` routine), defined in a header file `MC.h`:

```
double beam(double F, double L, double E, double I);

void MonteCarlo(int n, double E_mean, double E_stdev,
                Statistics& u_stat);
void run(int n);  // run MonteCarlo (kind of main program)
```

The implementations of these functions are listed next.

```
double beam(double F, double L, double E, double I)
{
  return F*L*L*L/(3*E*I);
}

void MonteCarlo(int n, double E_mean, double E_stdev,
                Statistics& u_stat)
{
  Statistics gauss_stat("gauss");
  for (int i=0; i<n; i++) {
    double E = gauss (E_mean, E_stdev);
    gauss_stat.add (E);
    u_stat.add (beam(1.0, 1.0, E, 1.0));
    if (((i+1) % (n/10)) == 0) {
      gauss_stat.report(std::cout); u_stat.report(std::cout);
    }
  }
}

void run(int n)
{
  double E_mean=2.0, E_stdev=0.2;
  setSeed (12374);
  Statistics u_stat("u");
  MonteCarlo (n, E_mean, E_stdev, u_stat);
  std::cout << "CPU time: " << double(clock())/CLOCKS_PER_SEC << std::endl;
  // n more simulations:
  MonteCarlo (n, E_mean, E_stdev, u_stat);
  std::cout << "CPU time: " << double(clock())/CLOCKS_PER_SEC << std::endl;
}
```

The main program is now very simple:

```
#include "MC.h"
#include <stdlib.h>
int main (int argc, const char* argv[])
{
  int n;
  if (argc == 2) { n = atoi(argv[1]); }
```

```
  else { n = 1000; }
  run(n);
}
```

Compiling the files yields an application that runs about 50 times faster than the plain Python version.

2.3.4 A NumPy Implementation

We could try to improve our plain Python script for the Monte Carlo simulation example by utilizing NumPy. The simple loop from 1 to the number of samples n must then be expressed by vector operations. This requires rewriting the `MonteCarlo` function. In particular, we need a mechanism for generating a long vector of random Gaussian numbers applying NumPy functions only. The latter task can be accomplished by first generating independent uniformly distributed numbers on the unit interval using the `RandomArray` module that comes with NumPy and then creating Gaussian variables in pairs (g_1, g_2) by the Box-Müller method:

$$g_1 = -2\ln u_1 \cos 2\pi u_2, \quad g_2 = -2\ln u_1 \sin 2\pi u_2,$$

where u_1 and u_2 are two independent uniformly distributed random variables on the unit interval. Here is a possible implementation in terms of NumPy array operations:

```
from Numeric import sqrt, log, sin, cos, concatenate, multiply
from RandomArray import random

def gaussian(n, mean, stdev):
    "draw n independent Gaussian random numbers"
    # u1, u2: independent uniformly distr. variables on [0,1]
    # use u1 and u2 for the sin and cosine part of the pairs resp.
    u1 = random(n/2)
    u2 = random(n/2)
    u1 = sqrt(-2.0*log(u1))
    u2 = 2*3.14159*u2
    s = sin(u2)
    c = cos(u2)
    a = concatenate((u1*s, u1*c))  # one long array
    # a is now N(0,1)
    multiply(a, stdev, a)
    return a + mean
```

The array-oriented version of `MonteCarlo` can take the following form, where we utilize two other features besides `Histogram` in the Scientific Python module: `average` and `standardDeviation` for computing the mean and the standard deviation of the numbers in an array.

```
def MonteCarloNumPy(n, mean, stdev, u_samples=None):
    "MonteCarlo calculations with NumPy vector calculations."
    import Scientific.Statistics
    import Scientific.Statistics.Histogram
    S = Scientific.Statistics  # abbreviation

    E = gaussian(n, mean, stdev)
    F = 1.0; L = 1.0; I = 1.0;
    u = beam(F,L,E,I)
    # add u to the end of previous samples:
    if u_samples == None:
        u_samples = u
    else:
```

```
        u_samples = concatenate((u, u_samples))

    print "E[u]=",     S.average(u_samples)
    print "StDev[u]=", S.standardDeviation(u_samples)

    # use an environment variable to turn on or off histogram
    # computation and plotting (since this is time consuming
    # and may produce less relevant CPU timings of the numerics)
    if os.environ.has_key('SKIP_HISTOGRAM'):
        skip_histogram = int(os.environ['SKIP_HISTOGRAM']) # 0 or 1
    else:
        skip_histogram = 0
    if not skip_histogram:
        h = S.Histogram.Histogram(u_samples,50);
        h.normalize()
        p = PlotThread(h.array,
            title='Histogram of deflection, n=%d' % (len(u_samples)))
        p.start()  # start the thread, i.e., call p.run()
    return u_samples
```

Observe that we work on vectors in all statements and that the simple beam function shown previously on page 19 also works with NumPy arrays as arguments (!). This is one example on the flexibility of scripting languages where we do not need to explicitly write the type of the function arguments.

A simple main program goes like this:

```
E_mean = 2.0;  E_stdev = 0.2
from RandomArray import seed
seed(1,2) # init with two integers
try:    n = int(sys.argv[1])
except: n = 10000
samples = MonteCarloNumPy(n, E_mean, E_stdev)
import time
print "CPU time:", time.clock()
# some more simulations:
samples = MonteCarloNumPy(n, E_mean, E_stdev, samples)
print "CPU time:", time.clock()
```

The script is found in `src/swig/sbeam/sbeam_numpy.py`. 200000 samples run at a speed about 15 times faster than the plain Python implementation, but still about three times slower than the pure C/C++ version. When histogram calculations and plotting are turned on, this NumPy-based script increases the CPU time by slightly more than a factor of two, quite independent of n.

2.3.5 Combining Python and C/C++

This section explains how we can combine the flexibility and convenience of Python scripts with the computational power of C/C++. What we would like to do is controlling the simulation in Python and do the number crunching in C and C++.

Before we start with the implementations of parts of our stochastic beam simulation example in C/C++, we should analyze where the bottlenecks are. An appropriate tool is to run the Python profiling tools on the plain Python script `sbeam_py.py`. Appropriate Unix commands are

```
export SKIP_HISTOGRAM=1
profiler.py sbeam_py.py 100000
```

The output table from `profiler.py` reads ($n = 100000$)

```
         1000180 function calls in 50.500 CPU seconds

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     2   19.030    9.515   50.450   25.225 sbeam_py.py:15(MonteCarlo)
200000   10.670    0.000   17.570    0.000 random.py:238(gauss)
400000   10.120    0.000   10.120    0.000 Statistics.py:19(add)
200000    6.900    0.000    6.900    0.000 whrandom.py:65(random)
```

Although most of the time is spent in the `MonteCarlo` function, the `random` module (which also calls `whrandom`) consumes 17 out of the 50 seconds in `MonteCarlo`, whereas the `Statistics` object consumes another 10 seconds. Hence, the obvious candidate for a quick improvement of the `sbeam_py.py` script is to rewrite the random number generation in C. A natural next step is to utilize the `Statistics` class in C++ and write the Monte Carlo loop in C++.

A profiling of the `sbeam_numpy.py` script is also of great interest. Running 1000000 samples gave

```
   124 function calls (122 primitive calls) in 9.880 CPU seconds

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
   6/4    5.410    0.902    5.410    1.353 __init__.py:14(moment)
     2    2.300    1.150    3.400    1.700 sbeam_numpy.py:15(gaussian)
     4    0.920    0.230    0.920    0.230 RandomArray.py:36
                                           (_build_random_array)
     2    0.760    0.380    0.760    0.380 sbeam_numpy.py:8(beam)
     3    0.370    0.123    0.370    0.123 Numeric.py:154(concatenate)
     1    0.030    0.030    9.870    9.870 sbeam_numpy.py:4(?)
```

Also here the random number generation (in the `gaussian` procedure) is time consuming. A possible improvement could be to be to operate on the NumPy arrays in our own C loops.

Based on the profiling results, four combinations of C/C++ and Python seem attractive in our stochastic beam simulation example:

1. Work with the pure Python script `sbeam_py.py`, but replace the Python random number generator by the `draw.c` routines in C.

2. Work with `sbeam_py.py`, but call a C++ function for the Monte Carlo loop, utilizing a random generator in C and the `Statistics` class in C++.

3. Work with the NumPy-based Python script `sbeam_numpy.py`, but with an implementation of the Monte Carlo loop and the generation of Gaussian random numbers in C. The C code must then operate directly on the C representation of NumPy arrays.

4. As point 3, but write the Monte Carlo loop and the `gaussian` function in Fortran 77.

It is an open question whether the last two tasks will really pay off, but the technique is of general interest, and the present application is well suited for a pedagogical introduction to computations on NumPy arrays in C/C++ and Fortran.

Wrapping Pure C Code

Working with SWIG is very easy when wrapping pure C code. Starting with building a Python interface to the functions defined in `draw.h` is therefore a good pedagogical example as well as a highly relevant strategy according to the profiling results. The

`draw.h` files defines three functions, which we want to call directly from Python scripts: `void setSeed(int)`, `double draw01()`, and `double gaussian(double,double)`. The recipe goes as follows.

Creating a Subdirectory for SWIG Files. Go to the directory contaning `draw.h` and `draw.c`, and create a subdirectory `swig`, or `swig-draw`, since we shall work with several extension modules now. Such a subdirectory is not required by SWIG, but it helps to keep a clean structure of C and SWIG files. Go to `swig-draw`.

SWIG Interface File. Create a file `draw.i` that specifies the interface that we want to wrap. In this case it should be all the functions declared in `draw.h`:

```
/* file: draw.i */
%module draw
%{
#include "draw.h"
%}

%include draw.h
```

Standard C comments can be used to put comments in `.i` SWIG files. The second line defines a module `draw`. This means that we can later import the C functions into Python by writing `import draw`. The code inside `%{ ... %}` should contain includes necessary to compile the specified interface, here `draw.h`. The `%include` line defines the functions we want to make a Python interface to. Here this is every function listed in `draw.h`. If only the `gaussian` function were of interest, we could instead replace the `%include` line by

```
double gaussian(double mean, double stdev);
```

The Python module `draw` would then only contain the `gaussian` function.

Running SWIG. Generate wrapper code by running

```
swig -python -I.. draw.i
```

SWIG can also generate Tcl and Perl interfaces; simply replace `-python` with `-perl5` to get a Perl interface. The `-I..` option tells `swig` to search for header files (like `draw.h`) in our parent directory. The `swig` command results in a file `draw_wrap.c` containing the C code that enables C functions to be called from Python. Without SWIG you would need to write such lengthy code yourself. How to write wrapper code is well explained in Beazley [5] or in van Rossum's Python Documentation (the part called "Extending and Embedding"). The reader is encouraged to scan these resources for getting a feeling of what is going on when calling C from Python.

Compiling and Linking. The compiling is done in the following steps:

1. run SWIG on the input file. This will generate wrapper code.
2. compile the C code and the wrapper code.
3. link the object files together to a shared library file.

This can be done with a Makefile or a shell script:

```
#!/bin/sh -x
# simple manual compilation of the draw module:
swig -python -I.. draw.i
PREFIX=`python -c 'import sys; print sys.prefix'`
VERSION=`python -c 'import sys; print sys.version[:3]'`
gcc -O -I.. -I$PREFIX/include/python$VERSION -c ../draw.c draw_wrap.c
gcc -shared -o draw.so draw.o draw_wrap.o
```

Testing the Module. First check that you have a shared library file `draw.so` or `drawmodule.so` Then invoke the Python interpreter and try

```
import draw
draw.setSeed(34)
draw.draw01()
draw.gaussian(0,1)
```

We can easily test the new `draw` module together with the `sbeam_py.py` script: Simply replace the call to `gauss` in the `random` module by `gaussian` from the new `draw` module:

```
from draw import gaussian
...
E = gaussian(E_mean, E_stdev)
```

Complete code is found in `sbeam_py_draw.py` in the directory

```
src/swig/sbeam/C/swig-draw.
```

The code runs at almost half the CPU time compared to the original `sbeam_py.py` script.

Making a Python Interface to the C++ Code

The natural way to improve the computational efficiency of the `sbeam_py.py` script, besides using NumPy arrays, consists in implementing all the computationally intensive parts in C or C++ code. Here we shall employ the C and C++ code from Chapter 2.3.3, *called directly from a Python script*. SWIG works very smoothly with C code, but integration with C++ code can be somewhat more challenging.

Moving the Deflection Function to the Draw Module. Just as an example on how easy it is to extend the `draw` module, we move the `beam` function from Python to C. The function can be typed directly in `draw.i` as follows:

```
%inline %{
  double beam(double F, double L, double E, double I)
    {
      return F*L*L*L/(3*E*I);
    }
%}
```

Running SWIG, as well as compiling and linking remain unchanged. The `inline` directive in SWIG specifies C code that we want to add to the module. That is, we do not need to put all the code in C files.

In the `sbeam_py_draw.py` code we replace

```
def beam(F, L, E, I):  return F*L*L*L/(3*E*I)
```

by

```
from draw import beam
```

The files resulting from this modification are available in the directory

```
src/swig/sbeam/C/swig-draw-beam.
```

The effect on the CPU of moving `beam` from Python to C is negligible, a fact that is not surprising from the profiling.

The SWIG Interface File. The new module is called `sbeam` and contains interface to the functions in `draw.c`, `MC.cpp`, and `Statistics.cpp` files. The function prototypes in `draw.h`, `MC.h`, and `Statistics.h` constitute the programming interface from a Python script, and contains enough information to generate the wrapper code. The definition of the interface to be wrapped by SWIG becomes as follows.

```
%module sbeam
%{
/* all includes required by C++ compilation of wrappers */
#include <draw.h>
#include <Statistics.h>
#include <MC.h>
#include <Python.h>               // Python as seen from C
#include <Numeric/arrayobject.h> // NumPy  as seen from C
%}

/* We want class Statistics to return its samples
(double* getSamples()) into a NumPy array in Python;
hence, we need NumPy support and automatic conversion
between C arrays and NumPy */

%init %{
    import_array()   /* initializing function for NumPy */
%}

%include "draw.h"
%include "MC.h"

/* Convert C array (double *) to NumPy array: */
%typemap(python, out) double* {
    int n;
    n = _arg0->getNoSamples();
    $target = PyArray_FromDimsAndData(1, &n,
                      PyArray_DOUBLE, (char *) $source);
}

%include "Statistics.h"
```

The complete interface file is `sbeam.i` in the `src/swig/sbeam/C/swig-MonteCarlo` directory.

The `typemap` facility in SWIG allows us to automatically convert the C array that is returned from `Statistics::getSamples` to a NumPy array. This functionality is very useful when we need to extract the samples from the `Statistics` object in C++ and pass the array on to the `Histogram` object, which expects the samples to be stored in a NumPy array.

Remarks.

1. If any of the files in the module are compiled with a C++ compiler, one needs to let C++ administer the linking process as well.

2. Using a C++ compiler for compiling all the files requires the `draw.h` to employ the `extern "C"` enclosure inside `#ifdef __cplusplus` directives (C++ compilers normally turn on the `__cplusplus` macro).

3. Make sure that the C and C++ code does not contain function names that coincide with Python keywords. This is the reason why the output function in class `Statistics` is called `report` and not `print`.

Running SWIG. Running SWIG with C++ code should be done like this:

```
swig -shadow -python -c++ -I.. sbeam.i
```

The `c++` option is required, whereas `-shadow` is optional but highly recommended. With `-shadow` SWIG makes so-called *shadow classes* for all C++ classes. Shadow classes are pure Python classes that behaves like the underlying C++ classes[4].

Compiling and Linking. The compilation process is the same as described in the "Compiling and Linking" section of Chapter 2.3.5, but we must use a C++ compiler for the C++ code and the linking. A shell script like

```
#!/bin/sh -x
swig -shadow -python -c++ -I.. sbeam.i
PREFIX=`python -c 'import sys; print sys.prefix'`
VERSION=`python -c 'import sys; print sys.version[:3]'`
gcc -fpic  -I..  -O2  -c ./../draw.c
g++ -fpic  -I..  -O2  -c ./../Statistics.cpp
g++ -fpic  -I..  -O2  -c ./../MC.cpp
g++ -fpic  -I..  -O2  -I$PREFIX/include/python$VERSION \
           -DHAVE_CONFIG_H -c ./sbeam_wrap.c
g++ -shared  -o sbeamcmodule.so draw.o Statistics.o MC.o \
                sbeam_wrap.o
```

compiles the module correctly.

Using the Module from Python. First we test that the new module work, e.g., on the fly in IDLE's Python shell:

```
>>> import sbeam
>>> n = 100
>>> u_stat = sbeam.Statistics("u", n)
>>> sbeam.MonteCarlo(n, 2.0, 0.2, u_stat)
```

Notice that all calculations are here done in the C/C++ code, but Python is used to control the program flow.

A new Monte Carlo function in Python can be written that takes advantage of the Monte Carlo function in C++, but adds histogram and plotting functionality:

[4]Without `-shadow` the Python interface to C++ classes is purely based on functions, not member functions of Python classes (see the SWIG manual for explanation).

```
def MonteCarlo(n, mean, stdev, u_stat):
    sbeam.MonteCarlo(n, mean, stdev, u_stat)
    #print "u_stats first 10 numbers: "
    #print u_stat.getSamples()[:10]

    # use an environment variable to turn on or off histogram
    # computation and plotting (since this is time consuming
    # and may produce less relevant CPU timings of the numerics)
    if os.environ.has_key('SKIP_HISTOGRAM'):
        skip_histogram = int(os.environ['SKIP_HISTOGRAM']) # 0 or 1
    else:
        skip_histogram = 0
    if not skip_histogram:
        from plot import plot, PlotThread
        from Scientific.Statistics.Histogram import Histogram
        h = Histogram(u_stat.getSamples(), 50)
        h.normalize() # let h be a density (unit area)
        title = 'Histogram of deflection, n=%d' % \
                u_stat.getNoSamples()
        import threading
        p = threading.Thread(target=plot, args=(h.array,title))
        p.start()
    return u_stat
```

A complete Python script is found in

```
src/swig/sbeam/C/swig-MonteCarlo/sbeam_py_MonteCarlo.py
```

Running this script with 200000 samples shows that it is about as fast as the pure C/C++ code.

2.3.6 Combining Python and Fortran

Python can call functions written in Fortran. A way to do this is to make a C interface to the Fortran code and then let Python call the C interface. Doing this manually is clearly an even more tedious and error-prone task than writing pure C extensions. Fortunately, there exists tools for writing wrapper functions automatically, somewhat like SWIG. We shall cover two tools, FPIG and Pyfort. Both tools support Fortran 77, and FPIG supports most Fortran 90/95 constructs.

How to wrap Fortran 77 code will here be demonstrated through the stochastic beam example from Chapter 2.3.1. We will do the same with FPIG and Pyfort for Fortran 77 code as we do with SWIG for C and C++ in Chapter 2.3.5. Our first task in this chapter is to write a random number generator module in Fortran and wrap the module to make it callable from Python's `sbeam_py.py` script.

A Draw Module in Fortran 77. There are no built-in random number generators in standard Fortran 77 so we need to provide suitable routines. The necessary subroutines make up a piece of code we shall refer to as the `draw` module. From Python we want to call two functions in this module: `setseed`, which initializes the random number generator, and `gaussian`, which calculates a random number drawn from the normal distribution with prescribed mean and standard deviation. The `setseed` and `gaussian` subroutines have the following signatures in Fortran 77:

```
subroutine setseed(seed)
integer seed
...
end
```

```
real*8 function gaussian(mean, stdev)
real*8 mean, stdev
...
return
end
```

In addition, the Fortran library contains a routine for generating uniformly distributed random numbers (called by `gaussian`). The complete source code is found in

```
src/sbeam/Fortran/draw.f.
```

Wrapping Fortran 77 Code With FPIG

We find it convenient to make a new subdirectory to separate the wrapper code from the Fortran code. The subdirectory is called `f2py-draw` when we use FPIG for creating a Python module `draw` based on Fortran 77 code. Generation of the files containing the wrapper code is performed with an `f2py` command on the form

```
f2py ../draw.f -m draw -h draw.pyf
```

The `f2py` program scans the Fortran code in `../draw.f`, writes the relevant interface specification in `draw.pyf`, and creates a makefile `Makefile-draw` for the `draw` module. By default, f2py generates wrappers for all the functions encountered in the Fortran files. In the present case we just want to create interfaces to two of the functions. This can be specified with the `only:` option, as in

```
f2py ../draw.f -m draw -h draw.pyf only: gaussian setseed \
     --overwrite-signature --overwrite-makefile
```

The two options `--overwrite-signature` and `--overwrite-makefile` are useful; otherwise f2py will not overwrite an existing interface file `draw.pyf` and makefile `Makefile-draw`. Of course, if you have made manual adjustments in these files, the overwrite options must be left out.

The interface file `draw.pyf`, generated by f2py, is actually the definition of a Fortran 95 module containing the functions we want in the interface. In the present example the generated `draw.pyf` file takes the form

```
python module draw ! in
    interface  ! in :draw
        subroutine setseed(seed) ! in :draw:../draw.f
            integer :: seed
            integer :: ncalls
            real*8 :: next_value
            common /gaussdraw/ ncalls,next_value
        end subroutine setseed
        function gaussian(mean,stdev) ! in :draw:../draw.f
            real*8 :: mean
            real*8 :: stdev
            integer optional :: ncalls=0
            real*8 optional :: next_value=0.0
            real*8 :: gaussian
            common /gaussdraw/ ncalls,next_value
        end function gaussian
    end interface
end python module draw
```

This file has enough information for f2py to create the correct wrapper functions. You can adjust the behavior of the Python interface to the Fortran routines by editing the `draw.pyf` file. For example, the function signatures in `draw.pyf` contains common block variables used in the functions. This information is not required. A minimal interface file is written manually in Chapter 2.3.6.

The next step is to generate wrapper code in C for the functions we want to access from Python. To this end, run `f2py` on the possible edited `draw.pyf` file:

```
f2py draw.pyf
```

All parameters needed for successful compilation and linking are included in the makefile `Makefile-draw`, so the creation of a shared library module, callable from Python, is accomplished by just typing

```
make -f Makefile-draw
```

You can now test that the module can be used from Python by running a one-line Python script, which just imports the new `draw` module:

```
python -c 'import draw'
```

A more detailed test is to write a script or invoke an interactive Python shell and call the `setseed` and `gaussian` functions:

```
import draw
draw.setseed(9862)
draw.gaussian(0,1)
```

Documentation of a Fortran function is available in the doc string. Say, for instance,

```
print draw.gaussian.__doc__
```

With exactly the same Python code as used when the `draw` module is implemented in C, `src/sbeam/C/swig-draw/sbeam_py_draw.py` we can now test the `draw` module. In other words, Python cannot distinguish whether the `draw` module is written in C or Fortran. The complete set of files for the Fortran 77 version of the `draw` module, including a copy of `sbeam_py_draw.py`, is found in `src/sbeam/Fortran/swig-draw`.

Wrapping Fortran 77 Code With Pyfort

Pyfort works in a way similar to FPIG; the only significant difference is that we have to write the interface file (like `draw.pyf`) manually. This is not very difficult if you are familiar with Fortran 90/95, but can be some work if there are many functions to wrap. The syntax of the interface files used by FPIG and Pyfort is almost identical since they both borrow the syntax from Fortran 95. A major difference is, nevertheless, that Pyfort does not use the `module` keyword. Actually, you can use FPIG to generate an interface for Pyfort if you remove the `module` keyword and its associated `end` mark.

We shall make a Python interface to the random number generation routines described on page 31. First we create a subdirectory `pyfort-draw` for the Pyfort files. The interface file, now written by hand, can look like

```
module draw
    subroutine setseed(seed)
        ! in :draw:../gaussian.f
        integer :: seed
        integer :: ncalls
        integer :: next_value
    end subroutine setseed

    function gauss(mean,stdev)
        ! in :draw:../gaussian.f
        real*8 :: mean
        real*8 :: stdev
        integer  :: ncalls
        real*8  :: next_value
        real*8 :: gauss
    end function gauss
end module draw
```

Notice that we limit the variables visible in this interface to subroutine arguments (FPIG automatically includes more variables in the interface).

Pyfort relies on Python's Distutils tool for compilation and linking of the new module. However, the Fortran files must be compiled into a library before running Pyfort. In the present case we could accomplish this by

```
g77 -O -c ../draw.f -o draw.o
ld -r -o libdraw_f77.a draw.o
```

Of course, `g77` can be replaced by any desired compiler. The next step is to run Pyfort:

```
pyfort -c g77 -m draw -b -L. -ldraw_f77 draw.pyf
```

The `-c` option specifies a compiler ID (not necessarily the name of the Fortran compiler, see the Pyfort manual), `-m` specifies the name of the module, `-b` denotes building the entire module without installing it, `-L` specifies the directory where the Fortran library (here `libdraw_f77.a`) is to be found, and `-l` is the name of the Fortran library without the leading `lib` and the suffix. Replacing `-b` by `-i` makes Pyfort install the library, i.e., copy `draw.so` into the standard Python libraries for modules such that you can use the new module easily from any directory.

The result of running Pyfort with the `-b` option is, hopefully, a directory tree `build` containing the wrapper code and the new shared library file `draw.so`.

The `draw` module can be tested by the following lines in a Python script or in an interactive Python shell:

```
import draw
draw.setseed(9862)
draw.gaussian(0,1)
```

Pyfort generates a file `draw.txt` with a documentation of the Python interface to the Fortran code[5].

The file `make.sh` in `src/sbeam/Fortran/pyfort-draw` lists all the statements needed to automatically build the `draw` module using Pyfort. The Python code `sbeam_py_draw.py`, making use of the random number generator in Fortran 77, can be found in the same directory and is identical to the FPIG and SWIG twins.

[5]Unfortunately, the text in this file is not available as doc strings in the various function objects in the Python code, which is the case when FPIG generates an interface.

2.4 Wrapping NumPy Arrays

We shall now return to the test scripts from Chapter 1.1.1, where we evaluate the function (1.1) at a large number of x values and place the results in an array. The efficiency of the script `fillarr1_NumPy.py`, based on arrays from NumPy, proved to be close to tailored C/C++ code. However, a disadvantage of this script is that we need to express the mathematical problem in terms of NumPy vector operations, which can be very difficult. Extensive memory requirements from temporary NumPy arrays in these vector operations constitute another potential problem in large-scale applications. It would hence be more convenient for many programmers and less memory demanding for the computer to make a direct loop over the NumPy array entries. Such a loop is extremely inefficient if it is coded in plain Python so we need to pass the NumPy array to C, C++, or Fortran code and perform the loop operations there.

NumPy arrays are represented by a C structure `PyArrayObject`. Its most important data members are

`int nd`
: The number of indices (dimensions) in the NumPy array.

`int *dimensions`
: Array of length `nd`, where `dimensions[0]` is the number of entries in the first index of the NumPy array, `dimensions[1]` is the number of entries in the second index, and so on.

`char *data`
: Pointer to the first data element of the NumPy array.

`int *strides`
: Array of length `nd` describing the number of bytes between two successive data elements for a fixed index. Suppose we have a two-dimensional (`PyArrayObject`) array `a` with `m` entries in the first index and `n` entries in the second one. Then `nd` is `m*n`, `dimensions[0]` is `m`, `dimensions[1]` is `n`, and entry `(i,j)` is accessed by

```
a->data-> + i*a->strides[0] + j*a->strides[1]
```

in C or C++.

2.4.1 Computing with NumPy Arrays in C

Suppose we want to implement evaluation of (1.1) in a C module where we work with NumPy's C data structure directly. We shall first make the Python-C interface from scratch, partly for showing how wrapper functions look like in C and partly for motivating the use of tools for automatic generation of wrapper code.

The Python documentation has a chapter "Extending and Embedding the Python Interpreter" (see [24]) that explains in detail how to call C functions from Python (Beazley's book [5] has a similar exposition). Using that information along with the Numeric documentation's, refer to [3] for guidelines on working with NumPy arrays in C, we can quite straightforwardly create the following C code. Although there is a lot of low-level administration of Python/NumPy data structures taking place in the function, the reader should from the comments be able to understand the basic structure and adapt it to similar problems. The relevant files are located in `src/ex/fillarr/NumPy/C/basic`.

```
#include <Python.h>              /* Python as seen from C */
#include <Numeric/arrayobject.h> /* NumPy  as seen from C */
#include <math.h>

double f (double x)
{
  double n, c, cH, A, c1, u;
  n = 0.3; c = 1.0 + 1/n; cH = pow(0.5,c);
  A = n/(1.0 + n);
  if (x <= 0.5) c1 = pow(0.5 - x,c);
  else          c1 = pow(x - 0.5,c);
  u = A*(cH-c1);
  return u;
}

static PyObject *fillarr_Numeric_C(PyObject *self, PyObject* args)
{
  PyArrayObject *array; /* C representation of Numeric array */
  int i,n;              /* number of array entries (x points) */
  double x;
  double* a;            /* C ptr to the NumPy array */

  /* parse the arguments to this function using Python tools */
  if (!PyArg_ParseTuple(args, "O!", &PyArray_Type, &array)) {
    /* extracting a NumPy array as argument was not successful */
    return NULL;  /* Error indicator */
  }

  /* check that we have a one-dimensional array */
  if (array->nd != 1) {
    /* throw a Python exception (ValueError) */
    PyErr_SetString(PyExc_ValueError,
                    "the NumPy array must be one-dimensional");
    return NULL;
  }
  /* check that the datatype is NumPy/Python float, i.e. C double */
  if (array->descr->type_num != PyArray_DOUBLE) {
    PyErr_SetString(PyExc_ValueError,
                    "the NumPy array must consist of floats");
    return NULL;
  }

  n = array->dimensions[0];    /* length of the array */
  a = (double*) array->data;   /* pointer to the NumPy data */

  /* what we came here to do: fill a with f(x) values */
  for (i=0; i<n; i++) {
    x = (double) i/(n-1);
    a[i] = f(x);
    /* more general: *(array->data+i*array->strides[0]) = f(x) */
  }

  /* return nothing; we just borrow a reference to the Python
     object */
  Py_INCREF(Py_None);  return Py_None;
}
/*
   the method table must always be present - it lists the
   functions that should be callable from Python:
*/
static PyMethodDef FillarrMethods[] = {
  {"fillarr_Numeric_C", /* name of func when called from Python */
   fillarr_Numeric_C,   /* corresponding C function */
   METH_VARARGS},
  {NULL, NULL}
};

void initfillarr_basic_C()
{
```

```
  /*
     assign the name of the module and the name of the
     method table:
  */
  (void) Py_InitModule("fillarr_basic_C", FillarrMethods);
  import_array();   /* NumPy initialization */
}
```

The type of the array elements is reflected by a `PyArray_Type` variable, accessed by the `descr->type_num` attribute of the NumPy structure. The Numerical Python manual lists the various legal data types. Computational scientists and engineers will make frequent use of `PyArray_DOUBLE` which means `double` in C, `float` in Python, and `Float` in NumPy.

We proceed with compiling the C file, here called `fillarr_basic_C.c`, and making a shared library out of it. A little Bourne shell script can perform the tasks:

```
#!/bin/sh
PREFIX=`python -c 'import sys; print sys.prefix'`
VERSION=`python -c 'import sys; print sys.version[:3]'`
gcc -O3 -I$PREFIX/include/python$VERSION \
    -c fillarr_basic_C.c -o fillarr_basic_C.o
gcc -shared fillarr_basic_C.o -o fillarr_basic_C.so
```

A little test script might read

```
import Numeric as N
n=20
a = N.zeros(n, N.Float)  # create an array with n zero entries
from fillarr_basic_C import *
fillarr_Numeric_C(a)     # fill a in a C function
print a
```

A complete timing script is found in `fillarr_NumPy_basic_C.py`. It runs about the same speed as the `fillarr1_NumPy.py` script, where we only use NumPy functions in Python.

Using SWIG to Simplify NumPy Operations in C

With SWIG we can automate the conversion from the Python representation of NumPy structures to plain C arrays. This is done using *typemaps*. The result is that the manual, low-level procedures like in the example above is simplified.

In our case a general typemap from a `PyArrayObject` to a `double` array looks like this:

```
%typemap(python,in) double * {
  PyArrayObject *py_arr;

  /* first check if it is a NumPy array */
  if (!PyArray_Check($source)) {
    PyErr_SetString(PyExc_TypeError, "Not a NumPy array");
    return NULL;
  }

  if (PyArray_ObjectType($source,0) != PyArray_DOUBLE) {
    PyErr_SetString(PyExc_ValueError,
                    "Array must be of type double");
    return NULL;
  }

  /* check that array is 1D and contiguous. */
  py_arr = PyArray_ContiguousFromObject($source,
                                        PyArray_DOUBLE, 1, 1);
```

```
  /* set _arg1 to the length of this array */
  _arg1 = py_arr->dimensions[0];

  /* set a double pointer to the NumPy allocated memory */
  $target = py_arr->data;
}
```

The typemap can be stored in a *SWIG library file*, e.g. `arraymaps.i` and can be included in any number of SWIG input files.

Having the typemap at our disposal, we can write a simple SWIG file where we put the `f(x)` function and the `fillarr_Numeric_C` function, with their contents exactly as it would have been in pure C without disturbing Python and NumPy data structures:

```
%{
double f (double x)
{
  /* implement f(x) formula... */
}
%}

%inline %{
void fillarr_Numeric_C (double *a, int n)
{
  int i;  double x;

  for (i=0; i<n; i++) {
    x = (double) i/(n-1);
    a[i] = f(x);
  }
}
%}
```

The C code can be inserted directly or with the `%include` statement. Our input file looks like

```
%module fillarr_swig_C

%{
#include <Python.h>
#include <Numeric/arrayobject.h>
#include <math.h>
%}

%init %{
   import_array()   /* Initialization function for NumPy */
%}

%include arraymaps.i

%{
double f (double x) { ... }
%}

%typemap(python, ignore) int n {
  $target = 1;
}

%inline %{
void fillarr_Numeric_C (double *a, int n) {
  int i;
  double x;

  for (i=0; i<n; i++) {
    x = (double) i/(n-1);
```

```
    a[i] = f(x);
  }
}
%}
```

The last little typemap tells swig to ignore the second argument in `fillarr_Numeric_C` from the Python side. That means we don't have to send the length of the array as the second argument, because the typemap finds it and sends it along to the C function. Running SWIG creates wrapper code that must be compiled and linked together with any C files to a shared library. On my computer (with Linux) this can be done as follows:

```
linux> swig -python fillarr_swig_C.i
Generating wrappers for Python
linux> gcc -O3 -c fillarr_swig_C_wrap.c -DHAVE_CONFIG_H \
-I/home/rogerha/ext/linux/include/python2.0
linux> gcc -shared fillarr_swig_C_wrap.o -o fillarr_swig_Cmodule.so
```

Usage from Python looks like

```
linux> python
Python 2.0 (#1, Jan 11 2001, 11:56:43)
[GCC 2.95.2 20000220 (Debian GNU/Linux)] on linux2
Type "copyright", "credits" or "license" for more information.
>>> import fillarr_swig_C
>>> import Numeric
>>> a = Numeric.zeros(10,'d')
>>> print a
[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
>>> fillarr_swig_C.fillarr_Numeric_C(a)
>>> print a
[ 0.          0.00759499  0.01055114  0.01134961  0.01144676  0.01144676
       0.01134961  0.01055114  0.00759499  0.        ]
```

In the same directory you can find a `Makefile` or an alternative script `make.sh` for compiling and linking the module. The script `fillarr_Numeric_swig_C.py` tests the module.

2.4.2 Computing with NumPy Arrays in Fortran

Working with NumPy arrays in Fortran code is trivial as soon as you have grasped the basics of using f2py or Pyfort from Chapter 2.3.6. You will see that NumPy arrays are much simpler to deal with in Fortran than in C (Chapter 2.4.1 treats the details of computing with NumPy arrays in C code).

Our first example concerns migrating the `fillarr.py` script from Chapter 1.1.1 to Fortran. Although we made a version of this script using NumPy functionality called from Python, see Chapter 1.1.1, it appears to be easier to implement the $f(x)$ function (1.1) and a loop over an array in straight Fortran 77 code. A suggested implementation looks as follows.

```
      PROGRAM FILLARR
      INTEGER N
      PARAMETER (N=100000)
      REAL*8 MYARRAY(N)

      DO 10 I=1,20
         CALL MAKELIST(MYARRAY, N)
10    CONTINUE
      END

      SUBROUTINE MAKELIST(MYARRAY, N)
```

```
      INTEGER N
      REAL*8 MYARRAY(N)
      INTEGER I
      REAL*8 X

      DO I = 0, N-1
         X = I/FLOAT(N-1)
         MYARRAY(I) = F(X)
      ENDDO
      RETURN
      END

      REAL FUNCTION F (X)
      REAL*8 X
      REAL*8 N, C, CH, A, C1

      N = 0.3
      C = 1.0 + 1/N
      CH = C**0.5
      A = N/(1+N)

      IF (X .LE. 0.5) THEN
         C1 = (0.5 - X)**C
      ELSE
         C1 = (X - 0.5)**C
      ENDIF

      F = A*(CH - C1)
      RETURN
      END
```

This code is found in `/src/ex/fillarr/C-F77/F77/fillarr.f`. Notice that Python interfaces will call `FILLARRF77` as `fillarrf77`, i.e., all names are normally translated to their lower-case equivalents (Fortran itself is case insensitive).

Remark. From the previous naming conventions in C versions of this code, it would be natural to replace `FILLARRF77` by `FILLARR_NUMPY_F77`. However, some Fortran 77 compilers, including the widely used `g77`, adds a double underscore to function and subroutine names already containing an underscore. This feature might be confusing when generating wrapper functions so we prefer to work with Fortran subroutine names without underscores.

Wrapping the Functions with FPIG

Making a Python interface to the function `FILLARRF77` in the `fillarr.f` file is trivial using FPIG. We generate the interface in the directory

```
src/ex/fillarr/NumPy/F77/f2py
```

The relevant f2py command is

```
f2py ../../../C-F77/F77/fillarr.f -h fillarr.pyf -m fillarr \
     only: fillarrf77 \
     --overwrite-signature --overwrite-makefile
```

The generated interface definition `fillarr.pyf` can be used as is so the next step is just compiling and linking, using the automatically generated makefile:

```
make -f Makefile-fillarr
```

The `fillarrf77` function in Fortran 77 takes two arguments: an array and the length of the array. A similar function in Python we would just require the array variable, as the length is an intrinsic part of the array object. Fortunately, the Python interface to Fortran routines behaves this way, that is, the length of the array is an optional argument, according to the interface definition in `fillarr.pyf`:

```
interface  ! in :fillarr
  subroutine fillarrf77(a,n)
    real*8 dimension(n) :: a
    integer optional,check(len(a)>=n),depend(a) :: n=len(a)
```

The specification of `n` as optional, with default value equal to the length of `a` allows the parameter to be omitted when calling `fillarrf77`. The following Python code segment demonstrates how the function can be called:

```
import fillarr
from Numeric import *
n = 10000
a = zeros(n,Float)
fillarr.fillarrf77(a)    # fill a
# alternative call:
fillarr.fillarrf77(a,n)  # fill a
print fillarr.f(0.76)  # evaluate f(x)
```

Writing `print fillarr.fillarrf77.__doc__` results in a documentation of the generated Python interface to `fillarrf77`, where the optional argument is clearly specified:

```
fillarrf77 - Function signature:
  fillarrf77(a,[n])
Required arguments:
  a : input rank-1 array('d') with bounds (n)
Optional arguments:
  n := len(a) input int
```

Multi-dimensional arrays can be handled in the same way, that is, the size of the array does not need to be provided as arguments in the call statement in the Python script. Consider, for instance,

```
  subroutine fill(a, m, n)
  integer m, n
  real*8 a(m,n)
      ...
```

An interface file could here contain the specification

```
python module ...
 interface
  subroutine fill(a,m,n)
   real*8 dimension(m,n) :: a
   integer optional,check(shape(a,1)==m),depend(a)::m=shape(a,1)
   integer optional,check(shape(a,0)==n),depend(a)::n=shape(a,0)
  end subroutine makearr
 end interface
end python module
```

The size of a multi-dimensional arrays in dimension `i-1` is computed as `shape(a,i)`. When the Fortran does code not apply unit base index in all dimensions, e.g.,

```
  subroutine fill(a, m, n)
  integer m, n
  real*8 a(-1:m,-n:n)
      ...
```

the interface file specifies `m` and `n` according to

```
subroutine fill(a,m,n)
 real*8 dimension(m + 2,2 * n + 1) :: a
 integer optional,check((shape(a,1)-2)==m),depend(a) &
        :: m=(shape(a,1)-2)
 integer optional,check((shape(a,0)-1)/(2)==n),depend(a) &
        :: n=(shape(a,0)-1)/(2)
end subroutine fill
```

That is, the array has zero base index when viewed from Python and the specified base indices `-1` and `-n` when viewed from Fortran. The Python interface moves all optional array size arguments to the end of the argument list. Therefore, it is recommended to do the same in the Fortran 77 code to ensure complete consistency between the original source and the Python interface. As a specific example, consider `subroutine fill2(m,a,n)`. Running f2py generates a Python interface `fill2(a,[m,n])`, where the `m` and `n` are optional and appear at the end of the argument list. Such reordering of the arguments can be confusing for newcomers to f2py.

The script `fillarr_NumPy_f77.py` shows how we run the CPU time test from a Python script, calling up the Fortran functions to do all the compuatations.

Wrapping the Functions with Pyfort

Pyfort requires us to provide an interface file. Very often, this file is identical or very similar to the interface file generated by f2py (except for the `module`–`end` and `interface`–`end` pairs of keywords). In the present case we tell Pyfort that we want to make a Python interface to the Fortran functions `fillarrf77` and `f`:

```
module fillarr
        subroutine makelist(myarray, n)
            integer n = size(myarray)
            real*8, intent(in):: myarray(n)
        end subroutine makelist
        function f(x)
            real*8 x
            real f
        end function f
end module fillarr
```

Notice that that the argument `n` has the length of the array as default value when `fillarrf77` is viewed from Python.

The next steps are identical to the ones described for creating the `draw` module (see page 40). We work in a directory, here `src/ex/fillarr/NumPy/F77/pyfort` separate from where the Fortran source code file is located. As usual with Pyfort, we need to compile the Fortran source into a library. Example of relevant commands are

```
g77 -O -c ../../../C-F77/F77/fillarr.f -o fillarr.o
ld -r -o libfillarr_f77.a fillarr.o
```

Thereafter we run Pyfort to generate the interface wrapper code, compile it, and link the new module:

```
pyfort -c g77 -m fillarr -b -L. -lfillarr_f77 fillarr.pyf
```

All the steps in creating the `fillarr` module are automated through the `make.sh` script. To check that the new module works, try for instance

```
from fillarr import fillarrf77
from Numeric import zeros
a = zeros(100,Float)
fillarrf77(a)
```

The script `fillarr_NumPy_f77.py` measures the CPU time of a number of calls to `fillarrf77`. The efficiency can be compared to the pure F77 code in `src/ex/fillarr/C-F77/F77`; the latter is slightly faster.

In this section we have seen both Pyfort and FPIG used. Both tools seem reliable and the wrapped code is both efficient and easy to use. However, FPIG can be more practical since interface files can be generated automatically.

2.5 Alternatives to Code Wrapping

Systems like CORBA, XML-RPC , and ILU are sometimes useful alternatives to the code wrapping scheme described above. The basic idea they build upon is running the target language code and application language code as separate processes, and handle communication between these processes. The processes don't even have to run on the same machines and the communication may go over network By obvious reasons there is more overhead in this approach, and for the use we are aiming for in this thesis such a strategy is slightly overkill.

Chapter 3

Problem Solving Environments

In the last chapter we saw several implementations of a relative simple example. When dealing with more advanced applications, communication with users must often be performed. This also applies for numerical applications, like simulators where input and output data must be handled correctly. We want to create *Problem Solving Environments* (PSE) for handling the user communication in numerical applications. By using a scripting language like Python we can create flexible PSE, and with mixed-language programming techniques we can create effective applications as well. A dynamic PSE makes it possible to do *computational steering*, i.e we can change conditions in the middle of a simulation. The goal is to create applications which are flexible, effective, modularized, and easy to extend.

In Chapter 3.1 we will introduce an example simulating water waves. The implementations will handle input data and produce output data which can be used for later representation. In Chapter 3.2 we enhance the simulator with better handling of input data, as well as processing of output data using XML, which is an excellent format for storing of structured data. An introduction to XML is given in Chapter 3.2.1 and usage of XML is shown in Chapters 3.2.2–3.2.3. Computational steering for the wave simulator is discussed in Chapter 3.3. Experience from wrapping large numerical libraries is outlined in chapter 3.4

3.1 A Wave Simulator

The idea of a Problem Solving Environments using scripting techniques is is most conveniently introduced through a non-trivial, specific example. The example to be used here concerns simulation and visualization of water waves. More specifically, we have a program which implements a numerical method for solving a partial differential equation describing *long water waves*[1]. Solving a wave equation is a classical problem in many sciences. We will derive a algorithm based on finite difference schemes to simulate the water waves. The problem is briefly described in the next section. More details regarding the mathematical model, the discretization, and numerics can be found in Chapter 1 of [16].

[1]Long water waves means that the wave length is much larger than the depth. Such water wave models are used for simulating storm surges, tides, swells in coastal region, and tsunamis. Tsunamis are destructive water waves generated by earthquakes, faulting, or slides, usually at the sea bottom. The waves travel at high speed (500 km/h may be a representative figure) over large ocean areas and may cause severe damage to the population during run-up on beaches.

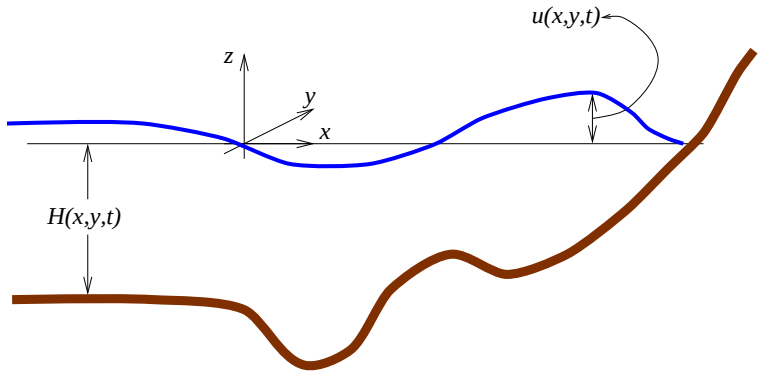

Figure 3.1: Sketch of long water waves, with the surface elevation and the bottom functions.

After the problem description we will show several implementations programmed in C, C++, Fortran, and Python. We will discuss the implementations abilities to be changed, extended, and of course the efficiency of the code. The implementations will range from a simple variant with little user communication to an implementation with a GUI and runtime computational steering features.

3.1.1 Problem Description

A possible mathematical model for long water waves consists of a *partial differential equation* (the so-called *damped wave equation* in this case)

$$\frac{\partial^2}{\partial t^2}u(x,t) + \beta\frac{\partial}{\partial t}u(x,t) = \frac{\partial}{\partial x}\left(H(x,t)\frac{\partial}{\partial x}u(x,t)\right) - \frac{\partial^2}{\partial t^2}H(x,t), \tag{3.1}$$

coupled with *initial* and *boundary conditions*:

$$u(x,0) = I(x), \tag{3.2}$$
$$\frac{\partial}{\partial t}u(x,0) = 0, \tag{3.3}$$
$$\frac{\partial}{\partial x}u(x,0) = 0, \tag{3.4}$$
$$\frac{\partial}{\partial x}u(x,L) = 0\,. \tag{3.5}$$

The function $u(x,t)$ models the elevation of the water surface, where $u = 0$ corresponds to still water (no waves), x is a spatial coordinate in the direction where the wave propagates, and t denotes time. See Figure 3.1 for a sketch.

The equation (3.1) is to be solved in a domain (fjord, lake, harbor, ocean basin) $[0, L]$. One sees that the domain is one-dimensional, while the water surface in the real world is two dimensional and the water motion is three dimensional. The assumption of *long* waves, relative to the depth, implies that the motion of the fluid particles in the vertical direction is so small that it can be neglected, resulting in a reduction in the number of space dimensions in the mathematical model. Here we also assume that the variation of the waves in one of the horizontal directions is negligible. This leaves us with a one-dimensional mathematical model describing two-dimensional water motion.

The function $H(x,t)$ in (3.1) models the still-water depth. Normally, H only varies with x, but the time dependence in H allows us to model slides along the bottom (which then results in a time-dependent bottom topography). The initial condition (3.2) says that the water surface at zero time has a shape according to the prescribed function $I(x)$. The other initial condition, equation (3.3), expresses that the water is at rest initially. The two boundary conditions (3.4) and (3.5) reflect that the wave does not penetrate the boundary at $x = 0$ and $x = L$, that is, there are reflective walls at $x = 0$ and $x = L$. This is a natural condition if we study waves in a fjord or lake with steep hills, but perhaps a less natural condition if the waves run up on a beach with small slope. Even in the latter case, (3.4) and (3.5) give results that are in sufficient agreement with observations of the nature. We should mention that the model (3.1)–(3.5) has been scaled such that all variables are dimensionless, the depth has a size of order unity (exact 1 if the depth is constant), and the domain (fjord, lake, harbor, ocean basin) has length L.

Of course, many physical effects are omitted in (3.1)–(3.5). We have already mentioned that the water surface is two-dimensional, demanding also another spatial coordinate and an extra term in (3.1). Moreover, the waves might be steep such that additional nonlinear terms should be included, and the waves might be too short for the underlying long-wave approximation[2]. Nevertheless, (3.1)–(3.5) is a sufficiently accurate model for many practical engineering and scientific studies of water waves.

The mathematical problem (3.1)–(3.5) must normally be solved by a *numerical method*. Here we apply the *finite difference method* on a uniform grid. Roughly speaking, the finite difference method consists in finding an approximation u_i^ℓ to $u(x,t)$ at discrete points in space and time, where i is a counter for spatial points and ℓ denotes a time level. Hence, at the third time level we have discrete values as depicted in Figure 3.2. The space between two grid points are called cells, and this discrete representation of the domain $[0, L]$ is called a *grid*.

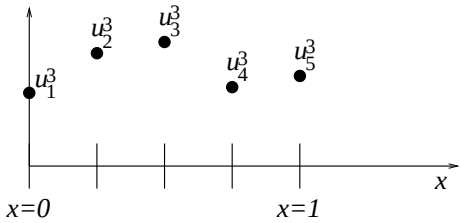

Figure 3.2: The discrete u_i^ℓ values as computed in a finite difference scheme with four cells in the domain $[0, 1]$.

The governing partial differential equation is assumed to hold at all the points in the grid at each time level. At each point, we approximate the derivatives in the mathematical model by finite differences:

$$\frac{1}{\Delta t^2}\left(u_i^{\ell-1} - 2u_i^\ell + u_i^{\ell+1}\right) + \frac{\beta}{2\Delta t}\left(u_i^{\ell+1} - u_i^{\ell-1}\right) =$$
$$\frac{1}{\Delta x^2}\left(H_{i+\frac{1}{2}}(u_{i+1}^\ell - u_i^\ell) + H_{i-\frac{1}{2}}(u_i^\ell - u_{i-1}^\ell)\right) - \frac{1}{\Delta t^2}\left(H_i^{\ell-1} - 2H_i^\ell + H_i^{\ell+1}\right)$$

[2]See [16, ch. 1 and 6] for details regarding modification of the mathematical model to account for the mentioned effects.

The quantity Δx is the spacing between the grid points, while Δt is the time lag between each time level, or in other words, the time step is Δt. This equation can be solved with respect to the new value $u_i^{\ell+1}$, yielding a so-called *explicit finite difference* scheme for $u_i^{\ell+1}$, where the values at time level ℓ and $\ell-1$ are considered as known. The function $H(x,t)$ is known and $H^\ell_{i+\frac{1}{2}}$ simply means to evaluate $H((i-\frac{1}{2})\Delta x, \ell\Delta t)$. However, in the code we use the arithmetic mean $H^\ell_{i+\frac{1}{2}} = \frac{1}{2}(H^\ell_i + H^\ell_{i+\frac{1}{2}})$.

The updating formula for $u_i^{\ell+1}$ becomes:

$$\begin{aligned} u_i^{\ell+1} &= \frac{2}{2+\beta\Delta t}\Big(2u_i^\ell + (\frac{1}{2}\beta\Delta t - 1)u_i^{\ell-1} + \\ &\quad \frac{\Delta t^2}{2\Delta x^2}((H_i^\ell + H_{i+1}^\ell)(u_{i+1}^\ell - u_i^\ell) + (H_{i-1}^\ell + H_i^\ell)(u_i^\ell - u_{i-1}^\ell)) - \\ &\quad H_i^{\ell-1} + 2H_i^\ell - H_i^{\ell+1}\Big) \end{aligned} \tag{3.6}$$

for all *internal* grid points $i = 2, \ldots, n-1$. Note that when u at time levels $\ell-1$ and ℓ are already computed, everything on the right-hand side of (3.6) is known. Refer to Langtangen's book ([16]) for more information about the derivation of the formulas.

At the boundary points, $i = 1$ and $i = n$, the boundary conditions (3.4)–(3.5) apply. Discretizing the the derivative gives

$$\frac{u_2^\ell - u_0^\ell}{2\Delta x} = 0, \quad \frac{u_{n+1}^\ell - u_{n-1}^\ell}{2\Delta x} = 0,$$

implying $u_2 = u_0$ and $u_{n+1} = u_{n-1}$. Using (3.6) for $i = 1$ and $i = n$ involves the fictitious values u_0 and u_{n+1} (outside the domain), but these can be replaced by u_2 and u_{n-1} because of the boundary conditions. The result is a special updating formula for the boundary points:

$$\begin{aligned} u_1^{\ell+1} &= \frac{2}{2+\beta\Delta t}\Big(2u_1^\ell + (\frac{1}{2}\beta\Delta t - 1)u_1^{\ell-1} + \\ &\quad \frac{\Delta t^2}{2\Delta x^2}((H_1^\ell + H_2^\ell)(u_2^\ell - u_1^\ell) + (H_2^\ell + H_1^\ell)(u_1^\ell - u_2^\ell)) - \\ &\quad H_1^{\ell-1} + 2H_1^\ell - H_1^{\ell+1}\Big), \end{aligned} \tag{3.7}$$

$$\begin{aligned} u_n^{\ell+1} &= \frac{2}{2+\beta\Delta t}\Big(2u_n^\ell + (\frac{1}{2}\beta\Delta t - 1)u_n^{\ell-1} + \\ &\quad \frac{\Delta t^2}{2\Delta x^2}((H_n^\ell + H_{n-1}^\ell)(u_{n-1}^\ell - u_n^\ell) + (H_{n-1}^\ell + H_n^\ell)(u_n^\ell - u_{n-1}^\ell)) - \\ &\quad H_n^{\ell-1} + 2H_n^\ell - H_n^{\ell+n}\Big). \end{aligned} \tag{3.8}$$

We have assumed that $H_0 = H_2$ and $H_{n+1} = H_{n-1}$, that is, $\partial H/\partial x = 0$ at the boundaries.

Initially, at $t = 0$, we have known $u_i^0 = I((i-1)\Delta x)$ and $\partial u/\partial t = 0$. The latter is discretized with a centered difference, $(u_i^1 - u_i^{-1})/(2\Delta t) = 0$, giving $u_i^1 = u_i^{-1}$. Applying

(3.6) for $\ell = 0$ and replacing the fictitious u_i^{-1} by u_i^1, we get a special scheme for u_i^1. At the boundaries, this scheme must be modified using $u_0 = u_2$, $u_{n+1} = u_{n-1}$, $H_0 = H_2$, and $H_{n+1} = H_{n-1}$. From a computational point of view it can be convenient to define u_i^{-1} such that (3.6)–(3.8) are valid also for the first time step ($\ell = 0$). The proper u_i^1 values are

$$\begin{aligned} u_i^{-1} &= \frac{1}{2}\Big(2u_i^0 + \frac{\Delta t^2}{2\Delta x^2}(H_i + H_{i+1})(u_{i+1} - u_i) \\ &\quad -(H_{i-1} + H_i)(u_i - u_{i-1}) - 2(H_i^1 - H_i^0)\Big), \quad i = 2, \ldots, n-1, \end{aligned} \tag{3.9}$$

$$\begin{aligned} u_1^{-1} &= \frac{1}{2}\Big(2u_1^0 + \frac{\Delta t^2}{2\Delta x^2}(H_1 + H_2)(u_2 - u_1) \\ &\quad -(H_2 + H_1)(u_1 - u_2) - 2(H_1^1 - H_1^0)\Big), \end{aligned} \tag{3.10}$$

$$\begin{aligned} u_n^{-1} &= \frac{1}{2}\Big(2u_n^0 + \frac{\Delta t^2}{2\Delta x^2}(H_n + H_{n-1})(u_{n-1} - u_n) \\ &\quad -(H_{n-1} + H_n)(u_n - u_{n-1}) - 2(H_n^1 - H_n^0)\Big). \end{aligned} \tag{3.11}$$

We have assumed that $H_i^{-1} = H_i^1$, i.e., $\partial H/\partial t = 0$ for $t = 0$. Physically, this means that the bottom is initially at rest.

The formula for u_i^1 involves the artificial quantity u_i^{-1}, which should have the following form to be compatible with the initial conditions (3.2)–(3.3):

$$u_i^{-1} = u_i + \frac{1}{2}\frac{\Delta t^2}{\Delta x^2}\left(H_{i+\frac{1}{2}}(u_{i+1}^0 - u_i^0) + H_{i-\frac{1}{2}}(u_i^0 - u_{i-1}^0)\right) \tag{3.12}$$

for all grid points $i = 1, \ldots, n$.

The Computational Algorithm

The computational scheme listed in the previous subsection can be expressed in a precise algorithmic form, a task to be accomplished before implementing the method in a computer program.

The computational scheme is only stable when Δt is sufficiently small,

$$\Delta t \leq \frac{h}{H_{\max}}. \tag{3.13}$$

In the implementation we choose

$$\Delta t = S\frac{h}{H_{\max}},$$

where S is a safety factor: $S \in (0, 1]$.

Algorithm 3.1

Simulation of waves.

define u_i^+ , u_i *and* u_i^- *to represent* $u_i^{\ell+1}$, u_i^ℓ *and* $u_i^{\ell-1}$, *respectively*
SET THE INITIAL CONDITIONS:
$u_i = I(x_i), \quad$ *for* $i = 1, \ldots, n$
DEFINE THE VALUE OF THE ARTIFICIAL QUANTITY u_i^- *:*
Equations (3.9)–(3.11)
$t = 0$
while time $t \leq t_{\text{stop}}$
  $t \leftarrow t + \Delta t$
  UPDATE ALL INNER POINTS:
  Equation (3.6)
  UPDATE BOUNDARY POINTS:
  Equations (3.7)–(3.8)
  INITIALIZE FOR NEXT STEP:
  $u_i^- = u_i, \quad u_i = u_i^+, \quad$ *for* $i = 1, \ldots, n$
  plot the solution (u_i, $i = 1, \ldots, n$)

Implementation notes

The development of a simulation model and its implementation in a simulator is normally carried out by a person who has extensive knowledge and experience with physics, mechanics, mathematics, numerics, and numerical programming. We have outlined the main points of the implementation in the previous text, and concluded with a algorithm for the problem and numerical method chosen.

When the algorithm is derived and understood it is quite easy to implement a simulator. But different programming languages have different features for doing massive array calculations. So some of the implementations will differ a bit from the original algorithm. Besides the implementations don't just implement the algorithm, but also plotting features and handling of user data and so on. How these parts of the simulators are implemented varies. We have implementations written in Fortran 77, C, C++ and Python. We will of course compare the computational speed, but also user communication and the possibility of runtime computational steering.

Input parameters. There are several parameters in the equations (3.6)–(3.11) and algorithm 3.1 that could be set to different values for tuning purposes. For maximum flexibility we want to give these to the simulator at run time, as command line parameters (an input file would be just as good). Here is a list of the available command-line options for a simulator:

```
-n            number of grid points, default 21
-tstop        stop time for simulation, default 10
-S            safety factor for time step calc., default 1
-m            magnification of u values(for plotting), default 1
-H            type of bottom function
-I            type of initial water elevation
```

The domain is fixed to be $[0, 10]$ (i.e. $L = 10$).

We can choose between several bottom types, i.e. `FlatBottom`, `ParabolicBottom`, `Slide1` and `BellBottom` which are valid choices for the `-H` switch. The `FlatBottom` type implements $H = 1$, whereas `ParabolicBottom` corresponds to the parabola $H(x) = x(10 - x)/25$. The `BellBottom` corresponds to a Gaussian bell function, that is, a sub sea hill:

$$H(x) = 1 - (1 - g_H) \exp\left(-\frac{(x - m_H)^2}{2\sigma_H^2}\right),$$

with σ_H and m_H as the standard deviation and the mean of the bell function and g_H as the minimum depth. These three parameters are set by the command-line arguments

```
-BellBottom-mean  -BellBottom-stdev  -BellBottom-gap
```

A possible function $H(x, t)$ for modeling an underwater slide is

$$\begin{aligned} H(x,t) &= \Delta - \beta(x + \epsilon)(x + \epsilon - L) \\ &- K\frac{1}{\sqrt{2\pi\gamma}} \exp\left(-\frac{1}{\gamma}\left[x - \frac{(L + \epsilon + 2)}{5} + ce^{\alpha t}\right]^2\right), \end{aligned} \tag{3.14}$$

where Δ, ϵ, L, K, γ, and α are constants that can be tuned to produce a particular slide. A suitable choice of values are $\Delta = 0.2$, $\beta = 0.04$, $\epsilon = 0.5$, $L = 11$, $K = 0.7$, $\gamma = 0.7$, $c = 1.2$, and $\alpha = -0.3$. This particular choice of H is called `Slide1` in the code. The `Slide1` function corresponds to (3.14). The various parameters in (3.14) can be assigned by the command-line arguments

```
-Slide1-Delta -Slide1-beta -Slide1-c -Slide1-K -Slide1-alpha
```

and so on.

Note that the simulators will not calculate $H(x)$ every time in the equations (3.6)–(3.11), but will calculate them once, and put the results in arrays. If the user choose a time dependent bottom type, the arrays will be updated for each time step. Note also that the points $H_{i+\frac{1}{2}}$ will be calculated with linear interpolation, $H_{i+\frac{1}{2}} = \frac{1}{2}(H_i + H_{i+1})$.

Just as for the bottom types we have some choices for the initial water elevation, which we also refer to as surface type. A trivial choice for `-I` is `FlatSurface`, corresponding to $I(x) = 0$. Other choices for `-I` cover `PlugSurface` with

$$I(x) = A \begin{cases} 0.5 - \pi^{-1}\tan^{-1}(\sigma(x - 5 - 2)), x > 5 \\ 0.5 + \pi^{-1}\tan^{-1}(\sigma(x - 5 + 2)), x \leq 5 \end{cases}$$

which is a plug shaped surface where σ controls the steepness of the plug (a large or moderate σ gives a steep plug, while a small σ gives a smoothing of the plug shape). This initial shape is interesting for studying the effect of numerical noise in finite difference methods [16, App. A.4.8]. The σ parameter is set by the command-line argument `-PlugSurface-sigma`.

Another choice of I is a Gaussian bell function, with the name `BellSurface`,

$$I(x) = \frac{1}{\sqrt{2\pi}\sigma_I} \exp\left(-\frac{(x - m_I)^2}{2\sigma_I^2}\right),$$

where σ_I and m_I are the standard deviation and the mean of the bell function, set by the command-line options

```
-BellSurface-mean  -BellSurface-stdev
```

respectively.

3.1.2 A Pure C Implementation

The goal of this implementation was to make the code as readable, user friendly, fast, and modular as possible. These goals are a bit contradictory but we have tried to reach a reasonable compromise. The application takes user instructions as input, runs the simulator with these parameters and saves the data for later visualization if asked.

User Communication. Some efforts have been made to give users ability to set important variables at run time. With old C compilers and Fortran code many of the variables and length of arrays would have to be set before compilation. Scanning of input parameters, and dynamic memory allocation makes it possible to simulate different situations without having to recompile the application.

Code Organization. The code is written in a procedural manner so the main routine just call some functions

```
int main (int argc, char **argv)
{
  struct Param p;
  struct DataStructure ds;

  scan(&p, argc, argv);     /* Read command line arguments. */
  ds = buildStructure(&p);  /* Allocate memory, etc.        */
  solveProblem(&p, &ds);    /* The solving routine.         */
  printResults(&ds);        /* Print some results.          */
  return 0;                 /* Sucsess!                     */
}
```

The `Param` and `DataStructure` structs holds all the variables and arrays, making the code more readable and easier to maintain.

Both Bottom and surface types are set at run time by using function pointers. Since the function pointers are inside structs this gives an almost object-oriented style. The bottom and surface type is set when reading the command line arguments

```
/* Check the (given) bottom function */
if (!strcmp(p->bf, "BellBottom")) {
    p->bottomFunc = bellBottom;
} else if (!strcmp(p->bf, "FlatBottom")) {
    p->bottomFunc = flatBottom;
} else if (!strcmp(p->bf, "ParabolicBottom")) {
    p->bottomFunc = parabolicBottom;
} else if (!strcmp(p->bf, "Slide1")) {
    p->bottomFunc = slide1;
} else {
    fprintf(stderr, "scan: Unknown bottom function: %s\n%s\n", p->bf, help);
    exit(1);
}

/* Check the (given) Initial function */
if (!strcmp(p->ic,"BellSurface")) {
    p->initFunc = bellSurface;
} else if (!strcmp(p->ic,"FlatSurface")) {
    p->initFunc = flatSurface;
} else if (!strcmp(p->ic,"PlugSurface")) {
    p->initFunc = plugSurface;
} else {
    fprintf(stderr, "scan: Unknown init function: %s\n%s\n", p->ic, help);
    exit(1);
}
```

The arrays are dynamically allocated at run time with size set by the user or the default value. This gives a fairly dynamical application, able to set variables and switch between different bottom and surface types with just a little effort. (We will see that this is much harder in Fortran 77, but quite easy in C++ and Python).

3.1.3 Python Implementations

In this section we will describe several implementations of the wave equation described in Chapter 3.1.1. The implementations will range from a version as simple as the C implementation in the section above, to one with a graphical user interface and computational steering features. A very interesting aspect is that Python modules and classes effectively enable code reuse. This fact together with Python's high level data structures makes it possible to write very compact code. In fact three complete Python applications has just a few more lines (about 10%) than one C implementation. We will describe the three implementations `wave1D`, `wave1DSteering`, and `guiWave`. For all these implementations we will have versions with only Python code, and versions with C code for the time critical parts.

A Naive Implementation

Our first Python implementation has the same features as the C implementation. Parameters, bottom type and initial surface type can be set at the command line. The code is organized as modules, where `Bottom`, `Surface` and `Wavesim1D` are the names of the modules. Most of the code is in the `Wavesim1D` module, which import the two other modules. Splitting the code in such modules gives more benefits than just making the code better organized and readable. If we decide to use another `Bottom` module, we can just replace the current one with the new or manipulate the so called `PYTHONPATH`[3]. We will make use of this technique later in this section

The `wave1D` module is organized as the C implementation in section 3.1.2 except that most of the functions belongs to a class. This is done for easy code usage reasons. The next more advanced implementations will be subclasses of this one. The code is organized as follows:

```
class Wavesim1D:
    def scan(self, argv):
        """
        Scan argv and set the variables that we need. Set either given
        or default value.
        """

    def report(self):
        "Report to the user info about grid, number of nodes, etc."

    def DHDu(self, i, ip, im):
        "Help function for calculating DHDu"

    def setInitCond(self):
        "Sets the initial conditions in the equation."

    def timeloop(self):
        "Run over the computational loop"

    def plotAtThisTimeStep(self, step):
```

[3]Modules in Python are searched for in a environment variable called PYTHONPATH on Unix and Linux systems. Inside Python, `sys.path` is a list with the same path.

```
            "Write arrays at this timestep to file"

        def solveAtThisTimeStep(self):
            "Solves the equation at the given timestep."

        def printResults(self):

        def plotResults(self):

        def solveProblem(self):
            "Scan args, set Intial Condition and solve."

    if __name__ == '__main__':
        simulator = Wavesim1D()
        simulator.solveProblem()
        simulator.printResults()
        if simulator.p2f:
            simulator.plotResults()
```

The `solveProblem` function looks pretty much the same as in the C implementation.

```
        def solveProblem(self):
            "Scan args, set Intial Condition and solve."

            self.scan(sys.argv[1:])
            self.report()
            self.setInitCond()
            self.timeloop()
```

The timeloop is also quite simple with just a loop over the timesteps containing calls to `solveAtThisTimeStep` and updating of arrays.

```
        def timeloop(self):
            "Run over the computational loop"

            n = self.nnodes
            step = 0
            while self.t < self.tstop:
                self.solveAtThisTimeStep()
                # Update for next timestep
                self.t += self.dt
                self.um = self.u
                self.u  = self.up
                self.up = self.um
                runup = self.u[0]
                if runup > self.max_runup: self.max_runup = runup

                # Update if bottom is time dependent
                if self.bf == "Slide1":
                    step += 1
                    self.Hm = self.H
                    self.H  = self.Hp
                    self.Hp = array(map(self.bottom.H, self.grid, [self.t]*n))

                # Print solutions to file?
                if self.p2f: self.plotAtThisTimeStep(step)
```

The actual calculations of the equations (3.6)–(3.8) is in the `solveAtThisTimeStep` function.

```
        def solveAtThisTimeStep(self):
            """
            Solves the equation at the given timestep.
            """
            n  = self.nnodes
            dt = self.dt
```

```
h  = self.h
beta = self.beta

# Update all inner points
for i in range(1,n-1):
    self.up[i] = (2*self.u[i] + (0.5*beta*dt - 1)*self.um[i] +
                  (dt/h)**2 * self.DHDu(i, i+1, i-1) -
                  self.Hp[i] + 2*self.H[i] - self.Hm[i])/(1 + beta)

# Left Boundary condition
if self.bcl == "dudn=0":
    i = 0;
    self.up[i] = (2*self.u[i] + (0.5*beta*dt - 1)*self.um[i] +
                  (dt/h)**2 * self.DHDu(i, i+1, i+1) -
                  self.Hp[i] + 2*self.H[i] - self.Hm[i])/(1+beta)
else:
    print "Error: solveProblem: radiation at x=0 not implemented\n"
    sys.exit(1);

# Right Boundary condition
i = n-1;
self.up[i] = (2*self.u[i] + (0.5*beta*dt - 1)*self.um[i] +
              (dt/h)**2 * self.DHDu(i, i-1, i-1) -
              self.Hp[i] + 2*self.H[i] - self.Hm[i])/(1+beta)
```

This Python implementation has more readable and better organized code than the C implementation, but the computational speed is much worse. Testing shows that it executes about 100 times slower than the C implementation, which makes this implementation unusable for large scale simulations. Fortunately we can improve the simulators efficiency, with only small changes in the code.

Efficiency Considerations

Before doing any code changes we use a profiler to find the bottlenecks in the simulator. When we have a time dependent bottom type like `Slide1` about 50% of the time is used in the `H` function (finding $H(x,t)$).

```
ncalls  tottime  percall  cumtime  percall filename:lineno(function)
354708   72.810    0.000   72.810    0.000 Bottom.py:62(H)
   705   30.310    0.043   50.940    0.072 wave.py:177(solveAtThisTimeStep)
353704   20.660    0.000   20.660    0.000 wave.py:78(DHDu)
     1   16.810   16.810  140.250  140.250 wave.py:128(timeloop)
```

When we have a time independent bottom type, the $H(x,t)$ values are the same for all timesteps, and the calculations are done only once. Then most of the time is used in the `solveAtThisTimeStep` and `DHDu` functions.

```
ncalls  tottime  percall  cumtime  percall filename:lineno(function)
   501   21.850    0.044   37.120    0.074 wave.py:177(solveAtThisTimeStep)
251500   15.310    0.000   15.310    0.000 wave.py:78(DHDu)
  1503    0.180    0.000    0.180    0.000 Bottom.py:32(H)
```

The obvious efficiency improvements must be to do something about the `H`, `DHDu` and `solveAtThisTimeStep` functions. First we try to use the abilities of the NumPy array, by replacing the inner loop with NumPy vector-oriented array operations. Thus we replace the inner loop in `solveAtThisTimeStep` function

```
# Update all inner points
for i in range(1,n-1):
    self.up[i] = (2*self.u[i] + (0.5*beta*dt - 1)*self.um[i] +
                  (dt/h)**2 * self.DHDu(i, i+1, i-1) -
                  self.Hp[i] + 2*self.H[i] - self.Hm[i])/(1 + beta)
```

with the code

```
# Update all inner points
dhdu =  array([0] + map(self.DHDu, range(1,n-1), range(2, n), range(n-2)) + [1])
self.up = (2*self.u + (0.5*beta*dt - 1)*self.um + (dt/h)**2 * dhdu -
           self.Hp + 2*self.H - self.Hm)/(1 + beta)
```

Here we create an array of the values from `DHDu` such that updating the `up` array according to the difference scheme can be done with vector operations. This improvement makes the code twice as fast, but unfortunately more difficult to read for many programmers (unless you are well known with vector-style operations). Besides the implementation is still 50 times slower than the C implementation.

Optimizing with C Extensions

It seems that the problem we want to simulate is to demanding to solve efficient with pure Python code[4]. So to reach further improvements we will use the code from the C implementation for the `H`, `DHDu` and `solveAtThisTimeStep` functions, and import them as modules into the Python code.

We follow the procedure described in more detail in Chapter 2 and wraps the C code in the necessary Python extension API code. The code for the `BellBottom.H` function looks like

```
static PyObject *
CBottom_Bell_H(PyObject *self, PyObject *args)
{
  double x, t, gap, mean, stdev;
  double result;

  if (PyArg_ParseTuple(args, "ddddd:Bell_H", &x, &t, &gap, &mean, &stdev)) {
    result = 1 - (1-gap)*exp(-pow(x-mean,2)/2*pow(stdev,2));
    return Py_BuildValue("d", result);
  }
  /* else ...*/
  PyErr_SetString(PyExc_TypeError, "Five doubles expected");
  return NULL;
}
```

The other `H` functions are written in the same manner. A really nice feature with this extension is that we can import it in our implementation without changing any code in the `Wavesim1D` module. All changes is within the `Bottom` module

```
class BellBottom(BottomFunc):
    def scan(self, argv):
        self.gap   = float(readComLineArg(argv, "-BellBottom-gap", "0.5"))
        self.mean  = float(readComLineArg(argv, "-BellBottom-mean", "5.0"))
        self.stdev = float(readComLineArg(argv, "-BellBottom-stdev", "0.5"))

        # Set variables in C extension
        CBottom.setBellVars(self.gap, self.mean, self.stdev)
        # Set BellBottoms H method to the C extensions Bell_H
        self.H = CBottom.Bell_H
```

Note that we set the variables as global variables (in the extension), such that the `H` functions points directly to the extension module functions. The alternative would be to call the functions like

[4]It should be mentioned that this example does not use neither Python or NumPy arrays to it's full potential. The computational algorithm are influenced by C/Fortran programming techniques. So other examples will probably be more fair to Python and NumPy, as the sbeam example in Chapter 2.3 shows.

```
class BellBottom(BottomFunc):
    def scan(self, argv):
        self.gap   = float(readComLineArg(argv, "-BellBottom-gap", "0.5"))
        self.mean  = float(readComLineArg(argv, "-BellBottom-mean", "5.0"))
        self.stdev = float(readComLineArg(argv, "-BellBottom-stdev", "0.5"))

    def H(self, x, t):
        return CBottom.Bell_H(x, t, self.gap, self.mean, self.stdev)
```

which is less efficient because of Python's function overhead. These changes makes the code four time faster when we have a time dependent bottom type, but still considerably slower than the C implementation (about 25 times).

Since no code is changes in the `Wavesim1D` we can just manipulate the `PYTHONPATH` environment variable to import the `Bottom` module from the extension directory, and then run the simulator as before. Another solution is of course to just import the new `Bottom` module and the other modules like this:

```
# Get Bottom and Surface classes
import sys
from Bottom import *

# Import Surface and wave from ..
sys.path = ['..'] + sys.path
from Surface import *
from wave import Wavesim1D

if __name__ == '__main__':
    simulator = Wavesim1D()
    simulator.solveProblem()
    simulator.printResults()
    if simulator.p2f:
        simulator.plotResults()
# end
```

The extension developed above don't really speed things up with the stationary bottom types. So we move on to write the whole `solveAtThisTimeStep` (with the `H` and `DHDu` functions) as an extension module. The idea is the same as above, to give a Python interface to the `solveAtThisTimeStep` and bottom functions and let them call any internal functions (like `DHDu`). As in the example above we set some global variables (global in the extension scope) at the scanning stage. In this implementation we must replace the `timeloop` function in the `wavesim1D` module to make use of the new `solveAtThisTimeStep` function. The `newWavesim1D` module looks like

```
class NewWave(Wavesim1D):
    def timeloop(self):
        "Run over the computational loop"

        n = self.nnodes
        step = 0

        # Help solveAtThisTimeStep extension  to chose right H function
        csolve.setBottom(self.bf, self.bcl)

        while self.t < self.tstop:
            csolve.solveAtThisTimeStep(n, self.dt, self.h, self.beta, self.up,
                                       self.u, self.um, self.Hp, self.H,
                                       self.Hm, self.grid, self.t)
            # Update for next timestep
            step += 1
            self.t += self.dt
            self.um = self.u
            self.u  = self.up
```

```
self.up = self.um
runup = self.u[0]
if runup > self.max_runup: self.max_runup = runup

# We don't need to update the H arrays if we don't have a
# time dependent Bottom function
if self.bf == "Slide1":
    self.Hm = self.H
    self.H  = self.Hp
    self.Hp = array(map(self.bottom.H, self.grid, [self.t]*n))

# Print solutions to file?
if self.p2f: self.plotAtThisTimeStep(step)
```

The module is a bit more complicated than the earlier versions, but still it's quite simple. And the use from Python is very simple. The only difference is that `solveAtThisTimeStep` must have more parameters.

Now we want to compare the efficiency for this implementation against the others. We see that this version is about three times slower than the C version. That is a fairly good result, considering that only small pieces of the code is rewritten in C. Besides all these test runs are done with plotting and file writing turned off, for the purpose of measuring the raw computational power. With the plotting turned on, all the programs will slow down, and these results will be less interesting. All results can be found in table 3.1.

# nodes	BottomType	C	Python	CBottom	CSolve
n = 501	BellBottom	0.12	12.90	12.79	0.38
n = 501	Slide1	0.73	77.18	20.23	2.67
n = 2001	BellBottom	1.78		201.1	2.80
n = 2001	Slide1	11.67			39.29

Table 3.1: Efficiency comparison of the different versions of the wave1D simulator. The pure C version is in the C column, the pure Python version in the Python column, the Python version with C bottom functions in the CBottom column, and the last Python version in the CSolve coloumn.

3.2 Enhancing the Wave Simulator

The implementation discussed in the section above does not really offer the user anything new. The source code is more modular and easier to read and extend, but for the user of the simulator there is little difference from the C implementation discussed in Chapter 3.1.2, except the efficiency. In this section we will implement new features improving user communication. First we will let the user give the input parameters to the equation in an XML format, as an alternative to the command line arguments. Second, the simulator will write the results to an XML file suitable for further processing.

3.2.1 Introduction to XML

XML is an acronym for *eXtensible Markup Language*, and was developed by the W^{3}C (*World Wide Web Consortium*)[5]. XML is ideal for storing so-called structured data. A

[5] Actually XML is still under development. XML version 1.0 was released in february 1998, and XML 1.0 second edition in october 2000. A new version and XML related standards are under development.

document can be split into elements by using XML. A major difference between XML and other markup languages is that tags (or more precisely elements) can be given names suitable for the content. E.g. the grid information for the 1D wave simulator in Chapter 3.1.1 can be stored in the XML format:

```
<grid start="0" stop="10">101</grid>.
```

The example contains a grid *element*, with two *attributes*; `start` and `stop`. The element data is the number of nodes in the grid.

As the example shows a document can be split into elements, where each element has a name describing the data content. If a document is extended with new types of data, one can mark them with a new element. We say that a document has a *logical structure* when it is split into elements. This simple but powerful idea makes XML a very flexible system for handling structured (or semi-structured) data and makes it usable for a wider range of purposes than just traditional markup. Another important issue is that XML is designed to be readable for both humans and computers. To give an element a name descibing its content makes it easier to read, and the logical structure makes the XML document parsable for computers. We will later see that an XML document has a tree structure, which makes fast computation possible. Software like XML parsers and XML tranformation engines are often called XML processors.

In the next sections we will briefly discuss traditional markup, how XML developed and the differences between XML and other markup languages as HTML and SGML. We will describe XML in more depth, and mention XML related technologies like XSLT, a powerful transformation language for XML documents. Furthermore we will discuss use of XML technologies for use in mathematical and technical application, and show some examples of this in the following sections.

Document Markup and Historical Context

Traditionally, markup have been used in typesetting systems for giving specific styles for text. Such systems used different techniques for tagging text blocks. Common for most of these systems are looking at text documents as data streams, i.e. sequential ordered characters. Thus the start and end of a block can be marked. The end mark has in some systems been marked by a *newline* character, or the start of a new markup tag. XML differs from this, since a start tag and end tag are both required. Before going into more details of XML we will look at the two markup languages which has been most influential to XML, that is SGML and HTML.

SGML is an acronym for *Standardized General Markup Language*, and was ratified as a standard by ISO in 1986. It was designed to be a general markup language for a wide range of applications, much like XML. But it never became a success for various reasons. Most important is that it was too advanced. It is quite a task to know and understand the whole SGML standard, and to develop stable, robust and effective SGML processors was a tremendous challenge. Most SGML software had to use a subset of SGML, and was still incomplete, ineffective and error-prone. Still we could say that SGML was a success from an academic point of view. Many of the ideas survived, and XML has inherited much from SGML.

Some years later, the world wide web (WWW) and a markup language which was called HTML (*HyperText Markup Language*) was developed. There were many people and institutions working on this, and Tim Berners-Lee and CERN had an active role in

the process. See e.g. Feizabadi's article in [2] for more information. The WWW and HTML became a vast success, at least if we count the number of HTML documents. But not everyone thinks of the HTML format as a success. HTML has serious limitations as a document format for using the web to its full potential. It lacks support for giving documents logical structure, making searching and automatic processing less effective. This is one of the reasons why XML was designed, to replace HTML as the document format on WWW.

The Element. The basic tool for markup in XML is the *element*. An element consist of a *start-tag*, a *end-tag* and the data enclosed by them, as we saw in the example above. A *start-tag* consist of the *element name* enclosed by `<` and `>`. The end tag consist of the element name enclosed by `</` and `>`.

Elements can contain other elements and data. Elements which only contains other elements are called *container elements*, and these contained elements are called *child elements*. Elements contained in the same elements are called *siblings*. Elements which contains data is said to have *data content*

```
<parent>
  <child>data</child>
  <child>More data</child>
</parent>
```

In this example `parent` is a container element, and the `child` elements are both siblings and have data content. Elements with both data and elements are said to have *mixed content*. An element which doesn't have content at all is said to be *empty*. A convenient feature is that empty elements can skip the end tag if the start tag ends with `/>` instead of the usual `>`.

The example above introduces *attributes*. The `grid` element above has two attributes `start` and `stop`. Attributes in XML must have a name and a value on the form: attribute name, equal sign and the attribute value enclosed by double or single quotes. An attribute is typically used for adding information about the element it belongs to. This is often very useful, especially if you don't want to assign this information to a child element. Attributes can hold about any character data, but it is wise to let attribute values be small pieces of information concerning the element. It is possible to narrow the allowed content of an attributes value. We will look more into this in chapter 3.2.1.

Document Structure and Design

It is important to distinguish between the physical and logical structure of a XML document. The physical structure is the order of data units in the document. The logical structure of a document is a set of rules saying something about the order and quantity of the data units in the document. These rules are defined in a DTD (*Document Type Definition*), if present. If a document refer to a DTD, an XML parser can use this DTD to *validate* the document, which means to check wether the documents strictly follows the rules set up in the DTD or not. All documents which is processed by an XML processor must be *well-formed* or else an error message is given. A short description of well-formedness is given below.

Physical Structure. A complete XML document should begin with a XML declaration like

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
```

This is a processing instruction for an XML processor indicating which XML version, the character set used, and indicates if other documents or a DTD is referred to. If a DTD is available, a declaration with the DTD should come next. This is called the doctype declaration, and could look like

```
<!DOCTYPE mydoc SYSTEM "mydoc.dtd">
```

After these two special declarations, *entity* declarations and ordinary elements can follow. Entities can be used for refering to files, or text blocks. This gives the possibility for making a framework XML file referring to other files for keeping a proper document structure. In simple cases you can of course let all the data be in one file. Notice that an entity is declared once, but can have any number of references to it.

Well-Formed XML Documents. There are certain specified rules which must be obeyed if an XML document should be called well-formed. These rules represent the minimum criteria necessary for XML processors to be able to read and process the documents. In fact, the XML standard instructs that XML processors should give an error message and stop the processing if a document is not well-formed[6]. We will give a short description of some of the rules below.

Non-empty elements must have start and end tags, but empty elements may skip the end tag if the start tag ends with `/>`. The names of the start and end tag of an element must be the same. Note that XML is case sensitive. There are rules for the element names as well, see "The XML Companion" [7] for more information.

Root element. A well-formed XML document must have a *root element* containing all other elements of the document.

Elements must not overlap. Elements may contain other elements and data, but can not overlap each other. Text like
`<title>Math is <emph>fun!</title></emph>`
is not allowed in XML documents. This should instead be written like
`<title>Math is <emph>fun!</emph></title>`.

Enclose attribute values in quotes. Both single and double quotes are allowed if the same type of quotes start and end the attribute value. Attribute values enclosed with single quotes may contain double quotes without character escaping of any sort.

Only use `<` and `&` to start tags and entities. XML assumes that and opening bracket always starts a tag, and that the ampersand always start an entity reference. These characters have predefined entities; `<` and `&` respectivly.

[6] This is very different from many HTML processors, like web many browsers, which allows malformed HTML and tries to fix the error.

Logical Structure. An important feature of XML is the possibility of using a formal set of rules to define document structure. This set of rules must be in the form of a DTD, a concept inherited from SGML. The use of a DTD is optional, but gives the opportunity to validate XML documents. This can be done with a validating XML parser, which will report if any errors is found.

When discussing the further topics we will create a DTD for input files to the wave simulator. Some exmples of XML input files will be shown too. This DTD will have a tree structure which reflects the document tree structure, but it is not identical. In a DTD elements are defined once, but in a document tree it may appear more than once (if allowed by the DTD).

First of all, a XML document should have a root element. The root element will typically have one or more child elements. In a input file we can call the root element "input". The other elements will contain data for the grid, the bottom type and other variables. An input file could look like

```
<input>
  <grid start="0" stop="20">101</grid>
  <bottom>
    <Bell>
      <mean>5.0</mean>
      <stdev>0.5</stdev>
    </Bell>
  </bottom>
</input>
```

Here, we set the start and end points, and the number of nodes for the grid. The `bottom` element set the type and some type dependent variables. A DTD that allows an element structure like this can look like:

```
<!ELEMENT input     (grid?, bottom?)>
<!ELEMENT grid      (#PCDATA)>
<!ATTLIST grid      start   CDATA  #FIXED "0"
                    stop    CDATA  "10">
<!ELEMENT bottom    (Flat | Bell | Slide1 | Parabolic)>
<!ELEMENT Bell      (gap?, mean?, stdev?)>
<!ELEMENT gap       (#PCDATA)>
<!ELEMENT mean      (#PCDATA)>
<!ELEMENT stdev     (#PCDATA)>
...
```

The DTD allow the `input` element to have two optional elements, that is `grid` and`bottom` respectivly. The first has two attributes and data content, while the latter contain one element setting the bottom type. A complete DTD is listed at page 64.

We have seen that elements can both contain other elements and free text (PCDATA). Elements that can contain other elements we say has a model group, consisting of those elements. We can decide the order of of the elements with two logic operators, the sequence connector ',' and the choice connector '|'. Both are used in our example above.

Quantity control is ensured by quantity indicators. If an element should occur once and only once, no further information is required. If an element is optional and cannot repeat, it must be followed by a question mark, '?'. Zero or more occurrences is ensured by an asterisk, '*'. If an element is required and may repeat, it must be followed by a plus sign, '+'. If more advanced structures are wanted, the operators must be combined

to gain the wanted effect[7].

As seen in our first example (at page 59) elements can have attributes. An attribute is defined by the an attribute declaration. These contains the name of the element which the attribute belongs to, the attribute name and the type of the attribute. The most common attribute types is the *CDATA* and the *name group*, which are those used in the examples below. The *CDATA* type is character data (or just simple text). The *name group* restricts values to one of a finite set. For example a name group like `(foo | bar)` specifies that the attribute with this name group should have one of those tokens as value. It is also possible to give attributes default values. There are many other types as well. For information about the attributes and attribute types, refer to the XML standard [9] or "The XML companion" [7].

XML Processing and Related Standards

There are more features of XML we could cover, but what is mentioned so far should be sufficient to give some understanding of the purpose and use of XML. On the other hand, there are a lot of standards and technologies related to XML which is important for effective use of XML. How to read, write and process XML documents is very important. Writing XML is a trivial matter for most applications, since XML documents are pure text. Almost any programming language can write text to files and thus create XML documents. Reading XML and do something useful with it is a far more complicated matter. This is a task for an XML parser. There is several parsers available, both free and commercial, implemented in several programming languages.

There are two fundamentally different approches to reading the content of an XML document. They are known as *event-driven* and *tree-manipulation* techniques. With the event-driven approach the document is processed in a strict sequence. The parser starts at the root node and goes from node to node. The tree-manipulation approach gives access to the entire document by building a node tree, which make it possible to go to any part of the tree and interrogate and manipulate in any order. We see that the tree-manipulation approach is very powerful, but it can be very memory demanding since a tree structure of the whole document is built. This could make this approach ineffective when handling large documents. The event-driven approach is less powerful, but much less demanding on memory and more effective.

For simple tasks like searching for specific elements and processing of its content, the event-driven approach is most effective. But sometimes this approach is to simple for the problem to solve. As an example the tree-manipulation approach is best when a full XML document should be converted to another format. The W^{3}C has develped a standard API for both approaches, ensuring that the same ideas and methods are common for all XML parsers. The Simple Api for XML (SAX) for event-driven parsing, and Document Object Model (DOM) for tree-manupilation respectivly.

We should also mention the XML Stylesheet Language Transformation (XSLT) standard, since we will use this below. XSLT is a very powerful language for transforming XML to other document formats. An XSLT processor uses the tree-manipulation approach, which gives the possibility for massive manipulation of the original node tree. Thus an XML document can be converted to another format with a totally different

[7]For some structures this is obviously hard to achieve. This is one of the reasons why the W^{3}C has developed an alternative to DTDs called XML Schema, which is more flexibel and has more control features.

structure.

Usage of XML Technologies

In Chapter 3.1 we have implementations of a wave simulator. This simulator can both take input parameters and write a lot of information from the simulation to a file or stdout. Both the input parameters and out-data could be structered as xml documents, giving some useful benefits. First, the input parameters should be correct and have proper values, which is easy to check with a validating xml parser. Second, it would be nice to convert (parts of) the out-data to one or more document formats. In an example we will convert all data to a HTML format, but only the most important data to a document format for printing to paper. The next sections gives examples of these ideas.

3.2.2 Verification of Input Data with XML

One advantage with sending the input parameters in an XML format is that verification of the input data is possible. With the implementations above it is possible to give wrong input parameters or parameters with no meaning in the specific context, e.g. the command

```
linux> wavesim1D -n 1000 -H Slide -BellBottom-gap 0.7.
```

There are two errors here, the first being a typo and the second has no meaning. The argument `-H Slide` is a typo since the bottom type is called `Slide1`. The argument `-BellBottom-gap 0.7` has no meaning, since it indicates that the `gap` parameter for `BellBottom` bottom type should be set to 0.7 even though we are not using that bottom type.

In the implementations discussed above these errors would just be ignored by the simulator, and default values would be set. This could be very frustrating if it result in wrong results or waste of computation time. We could of course improve the scanning routines and do full parsing of the input arguments, but that is not as simple as it may sound. In fact it is possible that the number of code lines doing the parsing would exceed the code lines of the rest of the application. With XML, existing parsers can be used and just an interface to the parser must be written. Another solution could be to adjust the input parameters format to suit the `getopt` module in Python or a similar module.

A DTD for the Input

For validation of the input data a set of rules must be given. These rules should decide which elements that are allowed, and their structure. A DTD is used for setting the rules, i.e. the logical structure of an XML document. The DTD sets the order, quantity and allowed content for the elements. We will list the DTD and explain the details below.

```
<!ELEMENT input     (grid?, tstop?, beta?, sfactor?, bcleft?, plot?,
                     bottom?, surface?)>

<!ELEMENT grid      (#PCDATA)>
<!ATTLIST grid      start   CDATA  #FIXED "0"
                    stop    CDATA  "10">

<!ELEMENT tstop     (#PCDATA)>
```

```
<!ELEMENT sfactor   (#PCDATA)>
<!ELEMENT bcleft    (#PCDATA)>

<!ELEMENT plot      (destination, casename?, magnification?)>
<!ELEMENT destination   (#PCDATA)>
<!ELEMENT casename      (#PCDATA)>
<!ELEMENT magnification (#PCDATA)>

<!ELEMENT bottom    (Flat | Bell | Slide1 | Parabolic)>
<!ELEMENT surface   (Flat | Bell | Plug)>

<!ELEMENT Flat      EMPTY>
<!ELEMENT Bell      (gap?, mean?, stdev?)>
<!ELEMENT Slide1    (stdev?, beta?, eps?, L?, K?, c?, alpha?, gamma?)>
<!ELEMENT Parabolic EMPTY>
<!ELEMENT Plug      (amplitude?, sigma?)>

<!ELEMENT mean      (#PCDATA)>
<!ELEMENT stdev     (#PCDATA)>
<!ELEMENT gap       (#PCDATA)>
<!ELEMENT amplitude (#PCDATA)>
<!ELEMENT sigma     (#PCDATA)>
<!ELEMENT beta      (#PCDATA)>
<!ELEMENT eps       (#PCDATA)>
<!ELEMENT L         (#PCDATA)>
<!ELEMENT K         (#PCDATA)>
<!ELEMENT c         (#PCDATA)>
<!ELEMENT alpha     (#PCDATA)>
<!ELEMENT gamma     (#PCDATA)>
```

An input file must have an input element which can contain zero or one of the elements `grid, ..., surface` in that order. We will, as in the previous implementations, let all variable have default values. Thus a input file may contain an empty input element (all though it would be simpler to not give a input file to the simulator). A grid element should contain the number of nodes, and start and stop values as attributes. The start values has a default value "0", which cannot be changed (see Section 3.1.1 to find the reason for this), and the stop value has a default value "10" which can be changed. A plot element can be given with child elements determining magnification, whether to plot directly or to files with certain filenames. Furthermore a bottom and surface element can be given, specifying type and possibly variable values. With this DTD you cannot give wrong variables to a bottom or surface type or mistype a variable, since the parser will find the error and stop the scanning. A full input file looks like:

```
<input>
  <grid stop="20">101</grid>
  <tstop>10</tstop>
  <beta>0.003</beta>
  <sfactor>1</sfactor>
  <bcleft>dudn=0</bcleft>
  <plot>
    <destination>file</destination>
    <casename>SIM</casename>
    <magnification>1</magnification>
  </plot>
  <bottom>
    <Bell>
      <gap>0.5</gap>
      <mean>5.0</mean>
      <stdev>0.5</stdev>
    </Bell>
  </bottom>
  <surface>
    <Plug>
      <amplitude>0.5</amplitude>
```

```
      <sigma>1000.0</sigma>
    </Plug>
  </surface>
</input>
```

While the DTD determines the logical structure, a parser may need to validate further. The data content of some elements should perhaps be of a certain type. The `grid` element above should have a integer as the data content. Such validation cannot be done with the DTD, and must be performed by the parser.

An Input Parser

The input parser should validate the input file according to the DTD, and build a suitable data structure of the input data. Since the whole input file should be parsed, and that it is relatively simple, makes the event-driven approach a good parsing strategy. We implement this by using the SAX interface, see page 63 of Chapter 3.2.1. For easy usage in the wave simulator we implement the parser as a module, which can be used directly from the `scan` function.

```
def scan(self, argv):
    """
    Scan argv and set the variables that we need. Set either given
    or default value.
    """
    import waveInputParser
    data = waveInputParser(argv)

    self.L      = int(data.grid['stop'])
    self.nnodes = int(data.grid['nnodes'])
    ...
```

The actual parser is written in a object-oriented fashion, following the SAX interface rules. This means that objects for handling document elements and errors must be created. Our parser has the following structure:

```
from xml.sax import saxlib
class DocumentHandler(saxlib.DocumentHandler):
     def __init__(self):

     def startElement(self,name,attrs):

     def endElement(self,name):

     def characters(self,data,start,length):

class ErrorHandler:
     def error(self, exception):

     def fatalError(self, exception):

     def warning(self, exception):

def parse(wave_input):
     app = DocumentHandler()
     err = ErrorHandler()

     import xml.sax.drivers.drv_xmlproc_val
     sp = xml.sax.drivers.drv_xmlproc_val.create_parser()
     sp.setDocumentHandler(app)
     sp.setErrorHandler(err)
     sp.parse(wave_input)
     return app
```

The interesting parts are the `startElement`, `endElement`, and `characters` functions. The `ErrorHandler` object just report and stop the parsing if errors occur, and the `parse` function is the driver function for this module. We let the parser go through all elements, and we create dictionaries for storing of the data; `grid`, `input`, `bottom`, and `surface` respectively. The elements which has attributes or contain other elements must be handled as special cases. For instance the bottom and surface elements can contain elements determining the type of bottom or surface.

```
def startElement(self,name,attrs):
    self.current_element_name  = name
    self.current_element_attrs = attrs
    # Handle Bottom and Surface
    if name == "bottom":
        self.inBottom = 1
    elif name == "surface":
        self.inSurface = 1
    elif self.inBottom and name in self.bottomTypes:
         # we got a correct bottom type
         self.bottom['type'] = name
    elif self.inSurface and name in self.surfaceTypes:
         # we got a correct surface type
         self.surface['type'] = name

def endElement(self,name):
    # Handle Bottom and surface
    if name == "bottom":
        self.inBottom = 0
    if name == "surface":
        self.inSurface = 0
```

The other elements are handled in the `characters` function, where we store the data. The elements are stored in dictionary with keys and values according to the element names and values. The `tstop` element, which can occur in the input file like `<tstop>10</tstop>` is put in the dictionary with 'tstop' as key and '10' as value. The specific code is listed below.

```
def characters(self,data,start,length):
    el = self.current_element_name
    at = self.current_element_attrs
    end = start+length
    if el == "grid":
        # get number of nodes
        self.grid['nnodes'] = data[start:end]
        # Get attributes
        self.grid.update(at.map)
    elif self.inBottom:
        self.bottom[el] = data[start:end]
    elif self.inSurface:
        self.surface[el] = data[start:end]
    else:
        self.input[el] = data[start:end]
```

The code of the parsing module and the simulator using it is found in the directory `src/wave/LongWave1D/python/`.

3.2.3 Processing of Output Data with XML

The wave simulators discussed above generates result and output data, where some is stored in files, some printed on the screen and plots are shown. Sometimes it is practical to store all this information for later usage. If the data is stored as XML, it is available

in a general storage format which can be converted to many other formats. It is worth to notice that others can use the data, and easily convert them to suit their needs. The conversion is best done by using XSLT, the standard for transformation of XML documents.

For a typical simulation of water waves it makes sense to store the values of the equation variables, the grid, the timestep and the other variables and results from the simulation. The results arrays can be quite large, and it may be wise to store them in separate (possibly binary) files. An output file can look like

```
<output>
  <setup>
    <grid stop="20">101</grid>
    <tstop>10</tstop>
    <beta>0.003</beta>
    <bottom type="BellBottom">...</bottom>
    <surface type="FlatSurface">...</surface>
    <dx>0.5</dx>
    <dt>0.07</dt>
    <ntsteps>2700</ntsteps>
  </setup>
  <plot>plot_0.07.ps</plot>
  <results>
    <maxdepth>2.3</maxdepth>
    <maxrunup>1.2</maxrunup>
    <cputime>1.28</cputime>
  </results>
</output>
```

Note that the output file will be generated by the simulator, and validation is not necessary for the conversion. Thus we do not need a DTD.

Conversion of XML to HTML

To show conversion of XML using XSLT, we provide an example of how the output file can be converted to HTML. Assume we want the result from the computation to be a part of another web page. Thus we may want to convert the xml code above to HTML code like:

```
<h2>A long water wave simulation</h2>

<img src="plot_0.07.ps"></img>
<p>The simulation took place with the following setup:</p>
<table>
<tr><td>Bottom Type</td><td>BellBottom</td></tr>
<tr><td>Number of nodes</td><td>101</td></tr>
</table>
...
```

The conversion is done by an XSLT processor, which must be given instructions for how the XML code should be converted. The instructions must be given in the XSLT language. A straightforward version for our example is

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="output">
    <h2>A long water wave simulation</h2>
    <xsl:apply-templates />
  </xsl:template>

  <xsl:template match="plot">
    <img>
```

```
      <xsl:attribute name="src"><xsl:value-of select="."/></xsl:attribute>
    </img>
  </xsl:template>

  <xsl:template match="setup">
    <p>The simulation took place with the following setup:</p>
    <table>
    <xsl:apply-templates />
    </table>
  </xsl:template>

  <xsl:template match="grid">
    <tr><td>Number of nodes</td><td><xsl:value-of select="."/></td></tr>
  </xsl:template>

  <xsl:template match="bottom">
    <tr><td>Bottom type</td><td><xsl:value-of select="@type"/></td></tr>
  </xsl:template>
  ...
</xsl:stylesheet>
```

What actually happens here is that the XSLT processor builds a node tree of the XML document. With the information from that tree, a new document can be created with the information that is relevant for the purpose. When much of the information should be passed from the XML document to the new one, it is convenient to convert the data while traversing the tree. The XSLT language may seem verbose, lengthy and clumsy, but it is very powerful when one gets the grasp of it. The data tree representing the XML document can be manipulated thoroughly, thus making it possible to generate a quite different document. An explanation of XSLT instructions follows.

First we go to the `output` node of the XML tree. That element has no useful data for the HTML document, and thus we move on to the child elements, after writing a `h2` header to the HTML document. The `<xsl:apply-templates />` means; move on the the child elements. for the `plot` element in the XML file we want to have a `img` element in the HTML file. This is done by telling the XSLT processor to put the data in the plot element (that is "plot_0.07.ps") in the `src` attribute of the `img` element. For the `setup` element we build a table in the HTML file, and let the information in the child elements fill the table cells, as we see for the `grid` and `bottom` elements.

In this section we have shown that an XML parser easily can be coupled with the wave simulator. This parser is used to parse and validate input data to the simulator. If the simulator writes the output in an XML format, we can use an XSLT processor to convert the output to a suitable format, such as HTML, PDF, and RTF. A simple example shows how HTML can be produced.

3.3 Computational Steering

Computational steering means to interactively steer or control a simulation. Real time plotting, analyzing, or parameter manipulation are examples of computational steering. This is a different approach than just set some input data to a simulator, start it, and study the results afterwards. Beazley's article [6] gives an explanation, motivation, and a large scale example of the issue. We will give an example of computational steering by adding more control of the simulations from the wave simulator introduced in the above sections. This is done in terms of user communication, real-time plotting features, and options for control of numerical variables. An important part of the computational steering aspect is the user interface, which gives the user of the application options for

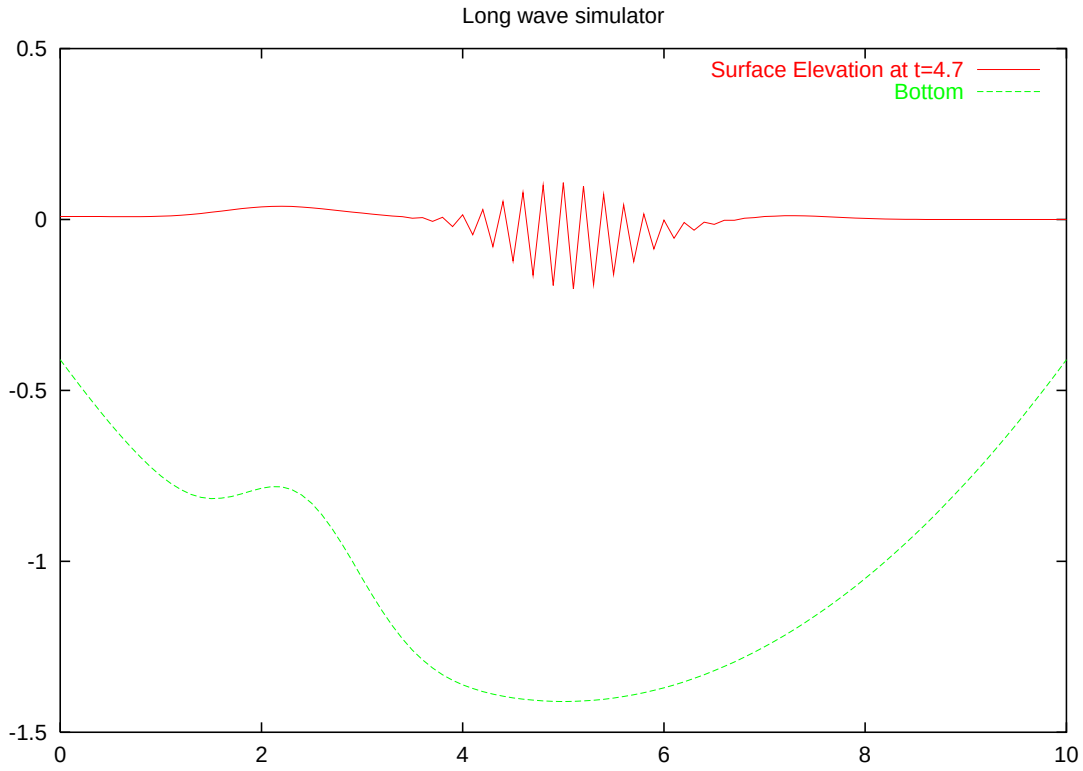

Figure 3.3: Wave simulation with numerical instability. A wave simulation which has been started with a too large value for the Δt variable. At timestep 4.6 the finite difference scheme becomes unstable due to Δt.

controlling the simulation. With a simple GUI this can be easy and intuitive. Before a description of the GUI is given, the computational steering example will be described.

In the problem description in Chapter 3.1.1, Δt is the time step variable of the difference scheme in equations 3.6–3.11. To avoid numerical instabilities that can arise from the difference scheme, Δt must satisfy the inequality 3.13. In short, this means that when the number of spatial nodes increase and the number of time steps must increase too, which means that Δt must decrease. This slows down the computation. It can be interesting to set Δt to a larger value than recommended, and adjust it if instabilities occur, which is what our example allows. Figure 3.3 shows a simulation where Δt has been increased with 30%, and almost half-way in the simulation, instabilities occur.

The GUI for the example described above must have the proper functionality. This includes options for halting the simulation, adjusting the Δt variable and restarting. In addition an option for "rewinding" the simulation to a time step before instabilities occurred must be offered. Because of this, an efficient module for array storage and lookup is used. Other features like save options for plots would be nice. All these features are offered in the example in the `src/wave/LongWave1D/python/CSolve/gui.py` application. The actual GUI is implemented using the `Tkinter` module in Python.

Much of the code from the previous version of the wave simulator is reused in this simulation. Some new code must be written for the GUI design and functionality, but only some of the control functions of the simulator are rewritten. The `setInitCond` must be rewritten, since plotting and array storage must be initialized.

```
def setInitCond(self):
    ...

    # Create NumPy array database
    self.array_store = NumPyDB.NumPyDBNumPyPickle(self.fname, 'store')

    # Plot window
    self.plot = Gnuplot.Gnuplot()
    self.plot('set data style lines')
    self.plot('set yrange [-1.5:0.5]')
```

The `timeloop` function must be rewritten too, because of the extra checking each time step.

```
def timeloop(self):
    n = self.nnodes
    while self.t < self.tstop and not self.halt_timeloop:
        self.plotAtThisTimeStep(self.grid, self.up*self.m, self.H*(-1), self.t)
        self.solveAtThisTimeStep()

        # Update for next timestep
        self.step += 1
        self.t += self.dt
        self.um = self.u
        self.u  = self.up
        self.up = self.um
        ...
```

The real time plotting functionality is done in

```
def plotAtThisTimeStep(self, grid, u, H, t):
    """
    Plot Surface and bottom at a timestep.
    """
    d1 = Gnuplot.Data(grid, u, with='lines',
                      title='Surface Elevation at t=\%2.1f' \% t)
    d2 = Gnuplot.Data(grid, H, with='lines', title='Bottom')
    self.plot.plot(d1, d2)
```

using Python's Gnuplot module. With this feature the plots from the simulation appears as a movie, though slow if the number of nodes is to large.

With the rewind functionality we are able to "play the movie" backwards if something unexpected happens in the simulation. If e.g. numerical instabilities occur, as shown in Figure 3.3, the simulation can be stopped, rewinded, Δt can be adjusted, and the simulation can continue. This means that when something goes wrong, the simulation must not necessarily start from scratch. This may not seem very useful in this example, but for simulations that run for hours or days, it can be really useful to stop, adjust, and continue. This is really the main point about computational steering, giving interactive control options to the user. A different strategy called `inverse steering`, is described in [13]. The idea of this strategy is to give feed the desired results of a computation to the simulator, and let it adjust the parameters necessary to achieve that result.

The efficiency for the computational steering example is somewhat lower. This versions run time grows with about 50 % compared to the standard Python version with file writing (for later plotting) turned on. Both versions use the fastest C-extension module, described on page 57. That means that this version, with computational plotting and real time plotting has a run time about five times slower than the pure C version. We observe that the computational steering model slows down the computations, but more freedom and flexibility is gained.

3.4 Wrapping Numerical Libraries

In the last two chapters we have wrapped many small pieces of target language code designed for the specific needs of some application. This is what we referred to as *mixed-language programming* in the introduction. Now we will show that it is possible to wrap large numerical libraries and use the functionalities offered from Python. A fortran library for hyperbolic partial differential equations called Clawpack, and a collection of C++ code for solving partial differential equations will be wrapped. Some simple applications in Python will show the usage for the wrapped libraries.

3.4.1 Wrapping CLAWPACK

CLAWPACK (Conservation LAWS PACKage) is a package of Fortran subroutines for solving time-dependent hyperbolic systems of partial differential equations in one, two, and three dimensions. This includes systems of conservations laws, but one can also solve nonconservative hyperbolic systems and systems with variable coefficients including source term. For more information about CLAWPACK, see [18].

To simplify the examples and reduce the amount of work, we will only consider one dimensional conservation law problems. That is equations on the form

$$\kappa(x)q_t + f(q)_x = \psi(q, x, \kappa), \tag{3.15}$$

where $q = q(x,t) \in \mathbf{R}^m$. The standard case of a homogeneous conservation law has $\kappa = 1$ and $\psi = 0$, giving the equation

$$q_t + f(q)_x = 0.$$

A simple version of this equation called the advection equation

$$q_t + uq_x = 0, \tag{3.16}$$

will be solved in our example below.

Wrapping the high-level routines

CLAWPACK is organized with subroutines in two levels. The high level routines provides an easy way to use CLAWPACK and is able to solve many problems. In one space dimension the routines available are `claw1ez` and `claw1`. Both these routines requires data files in a specific format and user-defined fortran routines. Thus, even if we wrap these high-level routines and access them from Python, some Fortran code must be provided too. This may seem strange, but a simple interface is made, and gives possibility to combine the application with other Python modules. If fortran programming is to be avoided, the low level functions must be wrapped. In that case the applications gain more flexibility, but more work must be done, as we will see in the next section.

First we will wrap the `claw1ez` subroutine with FPIG and test it from Python. The most important variables and arrays passed to this function are

`maxmx`: The maximum number of grid cells.

`meqn`: The number of equations in the hyperbolic system

`mwaves`: The number of waves produced in the Riemann solution

`mbc`: The number of "ghost cells" used for implementing boundary conditions

`work`: Work array with Dimension `maux`

`maux`: The number of "auxiliary" variables needed, i.e. the dimension of the array `aux`.

`q`: This array holds the approximate solution Q_i^n at a given time t_n.

`aux`: This array holds auxiliary variables used to define a specific problem.

The fortran code that must be provided is in the files

`qinit.f`: contains the subroutine `qinit` which sets the initial data in the array q.

`rp1.f`: contains the subroutine `rp1` which essentially is the Riemann solver for the advection equation.

`setprob.f`: contains the subroutine `setprob` which set problem-specific parameters.

In addition other types of boundary conditions can be set in a file `bc1.f`, but for our problem we will use the default boundary conditions.

We follow the recipe outlined in chapter 2.3.6, and use the command

```
f2py -m clawpy -h clawpy.pyf ../claw1ez.f
```

The auto-generated interface file turns out to be flawed, and must be corrected manually. The integer value of `aux` and the size and dimension of the array `maux` are set wrong by FPIG. The correct interface file looks like

```
python module clawpy ! in
  interface  ! in :clawpy
    subroutine claw1ez(maxmx,meqn,mwaves,mbc,maux,mwork,mthlim,q,work,aux)
      integer :: maxmx
      integer optional,check(shape(q,0)==meqn),depend(q)::meqn=shape(q,0)
      integer optional,check(len(mthlim)>=mwaves),depend(mthlim)::mwaves=len(mthlim)
      integer :: mbc
      integer :: maux
      integer optional,check(len(work)>=mwork),depend(work)::mwork=len(work)
      integer dimension(mwaves) :: mthlim
      double precision dimension(maxmx+mbc-(1-mbc)+1,meqn),depend(maxmx,mbc)::q
      double precision dimension(mwork) :: work
      double precision dimension(maxmx+mbc-(1-mbc)+1,maux),depend(maxmx,mbc)::aux
    end subroutine claw1ez
  end interface
end python module clawpy
```

The next step is to run `f2py` on the interface file:

```
f2py clawpy.pyf
```

and then compile the generated wrapper code and the fortran code, and link these with the CLAWPACK 1D library file. With this done we can test the new module with a simple python script:

```
import Numeric
import clawpy

maxmx = 500
mwork = 4032
mbc   = 2
meqn  = 1
mwaves = 1
maux  = 0

q      = Numeric.zeros((maux+1, maxmx+2*mbc), 'd')
aux    = Numeric.zeros(1, 'd')
work   = Numeric.zeros(mwork, 'd')
mthlim = Numeric.zeros(mwaves, 'd')

clawpy.claw1ez(maxmx, mbc, maux, mthlim, q, work, aux)
```

This little script solves equation (3.16), and the results can be plotted from the data files produced.

The `claw1` routine gives more flexibility than the `claw1ez` routine wrapped above, but the wrapping techniques will be the same. The same procedure can be followed to wrap the low level routines in the CLAWPACK 1D library.

3.4.2 Wrapping a C++ Library

Wrapping a C++ library is something quite different from wrapping a C or Fortran library. C++ has objects and offers more data types, which complicates the wrapping process. Thus, creating a functional and practical wrapper tool that handles all C++ features is very demanding. Thus many C++ wrapper systems use tighter code interfaces, i.e. the user must write a more detailed interface. This is a different strategy from what we see in e.g. SWIG or FPIG. Though, there are lot of available C++ code that are possible to wrap. For instance, the C++ code in the stochastic simulation example from Chapter 2.3 was wrapped with SWIG, without problems. In this section we will use the *Boost Python Library* to wrap parts of a more advanced C++ library, developed at the University of Oslo, see [22].

The C++ library contains classes for numerical arrays, grids, fields, and classes for a two-dimensional wave simulator. The most important parts of the `MyArray.h` are

```
template< typename T >
class MyArray
{
    T* A;                                   // the data
    int dimensions;                         // number of dimensions (max is 2)
    int length[2];                          // lengths of each dimension

    // needed for optimizations:
    int length0;
    int length0p1;

    // these three functions are used by constructors, destructors and redim:
    T* allocate(int length1);
    T* allocate(int length1, int length2);
    void deallocate();

    // transform fake indicies to real:
    int transform_index(int i, int j) const;

public:
    MyArray();
    MyArray(int n1);                  // constructor for 1D array
```

```
    MyArray(int n1, int n2);           // constructor for 2D array
    MyArray(const MyArray<T>& array);  // copy constructor
    ~MyArray();                        // destructor

    void redim(int n1);                // redim 1D array
    void redim(int n1, int n2);        // redim 2D array

    // return the size of the arrays dimensions
    int size() const;                  // length of 1D array
    int size(int dimension) const;

    bool indexOk(int i) const;         // check if index is ok to use
    bool indexOk(int i, int j) const;  // check if indicies are ok to use

    // operator() for 1D array
    const T& operator()(int i) const;
    T& operator()(int i);

    // operator() for 2D array
    const T& operator()(int i, int j) const;
    T& operator()(int i, int j);

    MyArray& operator=(const MyArray& v);  // assignment operator
                                           // (not implemented yet)
    // returns pointers to the data
    const T* getPtr() const;
    T* getPtr();

    void print(std::ostream& os);
};
```

The complete source code of `MyArray` and the other classes can be found in the directory `src/wave/Wave2D/C++`.

Now we will make a simple interface to `MyArray`, `TimePrm`, `GridLattice` and `Wave2D1`. The first step of the wrapping process is to write the wrapper interface. This interface must be written in C++, and uses the Boost library directly. This means that Boost header files are included and the library must be linked with the object files after compilation to create a python module (a shared library or DLL in most situations). A proper interface can look like

```
#include <Wave2D1.h>
#include <boost/python/class_builder.hpp>

// Python requires an exported function called init<module-name> in every
// extension module. This is where we build the module contents.
extern "C"
void inittest()
{
  try
    {
      // create an object representing this extension module
      boost::python::module_builder m("test");
      // Create a Python type object for our extension class (MyArray)
      boost::python::class_builder<MyArray<double> > m_a(m, "MyArray");
      // Add the __init__ function
      m_a.def(boost::python::constructor<>());
      m_a.def(boost::python::constructor<int>());
      m_a.def(boost::python::constructor<int, int>());
      // Add a regular member function
      // m_a.def(&MyArray<double>::print, "print");
      // Overloaded member functions
      m_a.def((void (MyArray<double>::*)(int))
              &MyArray<double>::redim, "redim");
      m_a.def((void (MyArray<double>::*)(int, int))
              &MyArray<double>::redim, "redim");
```

```
        m_a.def((bool (MyArray<double>::*)(int) const)
                &MyArray<double>::indexOk, "indexOk");
        m_a.def((bool (MyArray<double>::*)(int, int) const)
                &MyArray<double>::indexOk, "indexOk");
        m_a.def((int (MyArray<double>::*)(void) const)
                &MyArray<double>::size, "size");
        m_a.def((int (MyArray<double>::*)(int) const)
                &MyArray<double>::size, "size");

        // Create a Python type object for our extension class (TimePrm)
        boost::python::class_builder<TimePrm> time_class(m, "TimePrm");
        time_class.def(boost::python::constructor<double, double, double>());

        // Create a Python type object for our extension class (GridLattice)
        boost::python::class_builder<GridLattice> grid_class(m, "GridLattice");
        grid_class.def(boost::python::constructor<>());

        // Create a Python type object for our extension class (Wave2D1)
        boost::python::class_builder<Wave2D1> wave_class(m, "Wave2D1");
        wave_class.def(boost::python::constructor<>());
        wave_class.def(&Wave2D1::scan, "scan");
        wave_class.def(&Wave2D1::solveProblem, "solveProblem");
      }
    catch(...)
      {
        boost::python::handle_exception();    // Deal with the exception for Python
      }
}
```

The interface file is an abstract layer over the Python C extension API. For instance, the line

```
        boost::python::module_builder m("test");
```

creates the Python module `test`. The equivalent way to do this in the C extension API is with code like

```
static PyMethodDef testMethods[] = {
  /* Func_name_from_python, func_name_in_C, */
  {"test_foo", foo, METH_VARARGS},
  {NULL, NULL}
};

void inittest()
{
  (void) Py_InitModule("test", testMethods);
  import_array();   /* NumPy initialization */
}
```

where `testMethods` is the function table, mapping function names from C to the Python module. The way to interface the MyArray class and some of its methods is

```
        boost::python::class_builder<MyArray<double> > m_a(m, "MyArray");
        m_a.def(boost::python::constructor<>());
```

Here we make `MyArray` available from Python, with a constructor that takes no arguments. The other classes and functions are wrapped in a comparable way. The complete interface file and compile instructions is located in `src/wave/Wave2D/C++/bpl-MyArray`.

The wrapped classes and methods can be tested interactively from Python

```
>>> import test
>>> a = test.MyArray()
>>> a.redim(10)
```

```
>>> print a.size()
10
>>> a.redim(10,2)
>>> print a.size(1)
10
>>> print a.size(2)
2
>>> g = test.GridLattice()
>>> print g
<GridLattice object at 0x8385c28>
>>> sim = test.Wave2D1()
Hi!
```

Unfortunately, wrapping all the classes in the library are both difficult and time consuming. Thus, no useful Python applications interfacing the library code is shown.

Chapter 4

Conclusions and Final Remarks

In the introduction we showed how the efficiency for some scripting languages suffers when doing numeric calculations in explicit loops. For Python, however, we showed that using the Numeric module we could almost reach the speed of C and Fortran for our simple example. Other scripting languages have modules or techniques for numeric computing as well.

In Chapter 2 we explained how it is possible to interface and use code written in another language from most scripting languages. Python is excellent in the role of application language for numerical purposes, and C and Fortran 77 are good choices for languages to be interfaced from Python, i.e. as target languages. C++ is less suitable as target language, mostly because of its complexity. Various tools for wrapping C, C++ and Fortran code to Python are described, and some are used in the main example of the chapter. Among the available wrapping software, we have used SWIG and FPIG most extensively, and have found them to be very adequate and mature. Also Pyfort works well for wrapping Fortran 77 code.

The main example of Chapter 2 deals with stochastic simulation and implementations using C, C++, Fortran and Python are described. Benchmarks show that a pure C/C++ version is 50 times faster than the Python version, and three times faster than the NumPy version, i.e. the Python version using the Numeric module. Finally a Python version interfacing some of the C/C++ code is as fast as the pure C/C++ version. In addition the Python implementations have optional plotting and statistical features, which increase their utility. Interfacing Fortran code from Python yields the same efficiency as the corresponding C code. The example from Chapter 1.1 is now implemented with C and Fortran modules in Python, with the same resulting efficiency as when implemented purely in C or Fortran.

A more complicated example is introduced in Chapter 3, where we study a wave simulator. With this example we focus on user communication and computational steering in addition to efficiency. A simple C implementation tuned for speed is about a hundred times faster than a pure Python version, and three times faster than the most efficient extended Python version. This shows that a python application with a simple extension module not always compare to the speed of C, which may happen when the program gets complicated or there are several parts of the program using much CPU time. In such situations a better and more efficient module can be implemented, but that can be quite time consuming. On the other hand, the Python implementations can easily be coupled with an XML parser for better input and output handling and a GUI and plotting tools for better user communication and computational steering. This is shown

in Chapter 3.2 where XML is used for parsing of input data, and conversion of output data. Python's XML modules are handy for doing parsing and verification of input data to the wave simulator, and processing of output data to a wanted format. Run-time steering of the Δt variable is performed in Chapter 3.3. Introducing run-time steering naturally slows down the application due to the unavoidable extra checking that must be done every n'th time step. The flexibility from computational steering in this example necessarily results in a slower application.

The experience from wrapping larger libraries written in Fortran 77 and C++ is outlined in Chapter 3.4. Parts of the Fortran library *Clawpack* are wrapped and used successfully. But due to time limitations only some of the functions for the one-dimensional library of Clawpack are wrapped. This limits the utility of the module, but should be sufficient to show that also larger numerical Fortran libraries can be used from Python. Wrapping C++ code turned out to be more difficult, as expected. Simple C++ code like the code from the main example of Chapter 2 poses no problems, but when for instance more advanced features like those found in the Standard Template Library are used, trouble often arises. The local numeric C++ library was wrapped with only limited success with any of the packages describes in Chapter 2.2.3. However, both the Boost Python Library (BPL) and Siloon look promising, and may become more mature in the future. But for now, wrapping often requires great knowledge of C++, the code to be wrapped, and the wrapper tool, in addition to much work.

The experience gained from the examples and the work with this thesis has proved that Python in combination with C or Fortran can be e very good solution for numerical applications. The C or Fortran code can range from small pieces of code designed for a specific application to a more general library. Especially Python and Fortran can be an excellent choice, for several reasons. Fortran is generally recognized as the fastest language for numerical computations. The availability of very good tools for wrapping of Fortran code for use from Python is an important issue. Both FPIG and Pyfort are able to automatically convert Fortran arrays to NumPy arrays. Another very important issue is that Fortran is the by far most commonly used programming language for numerical and high-performance computing. There is a huge amount of applications and libraries available for almost any kind of numerical fields, and much of this code has seen heavy use in production environments for many years. It must therefore be considered to be among the most reliable numerical code in existence. This code can be reused in Python modules and connected to new software. The combination of Python and C can also be a great choice. The most common implementation of Python supports C directly through the API described in [24], and tools like SWIG simplify the process of wrapping C code a lot. Numerical C code is also very fast, sometimes as fast as Fortran code. On the other hand, wrapping C code can be more challenging than wrapping Fortran code, as C is a larger and more high-level language than Fortran. One should also be aware that less numerical software has been written in C than in Fortran. The average C programmer will therefore have to write and debug more of his software by himself than the average Fortran programmer.

For numerical applications, the combination of a high-level scripting language like Python and an efficient low level language like C or Fortran is an option very much worth considering. This combination lets one prioritize relatively freely between run-time efficiency and development time expenditure. There are even good chances that one can reduce development time without compromising run-time efficiency at all. If numerical code useful in the project already exists, development time may be cut even

further. Python paired with Fortran or C is a good alternative to modern object-oriented languages like C++, Java and Fortran 95, and also compares favorably with programming environments like Matlab and Octave.

4.1 Further work

There are several projects which can build on the work done in this thesis. Some are in proceeding by the author, but are in an early state and it is to time demanding to present the results here. Some of the examples in the previous chapters are dealing with partial differential equations (PDE). Creating useful and general modules for handling some types of PDE using finite difference methods or finite element methods based on the methods described in this thesis could be an interesting project.

We have learned that it is a very demanding task to make interfaces to C++ code from Python, as well as other application languages. There are wrapper tools developed for this purpose, but none of them are as good as e.g. FPIG, yet. A challenging project could be to create or participate in developing such software.

Other application languages could be worth considering. Common Lisp which first version appeared already in 1956 could be very interesting. The main reason is that Common Lisp is very flexible. It can be interpreted like Python, but there are good compilers too which can create efficient machine code. Another important fact is that Common Lisp in general is dynamically typed, as Python, but static typing for some variables can be offered as well. This means that for efficiency reasons a variable can be set to be a two-dimensional array, and other variables can be dynamically typed in the same application. This flexibility makes Common Lisp extremely interesting as an application language for numerical purposes. There exist good tools for interfacing Fortran and C code for most Common Lisp implementations. Unfortunately Common Lisp is rarely used in high-performance computing, and the Common Lisp implementations that are considered best are commercial software.

A new scripting language called Ruby could be an interesting application language. It has many similarities with Python and Perl, which applies both to syntax and intended use. Ruby can be extended with C and C++ code, but the author is not aware of tools for wrapping Fortran code. An interesting aspect of Ruby is that it is created to be easily extended. As an example the run-time garbage collection system in Ruby are more advanced than in e.g. Perl and Python. The garbage collection system is implemented with a so called *mark and sweep* procedure, which means that extension modules do not need to take care of reference counting, like in Perl and Python.

Bibliography

[1] David Abrahams. *The Boost Python Library*. See http://www.boost.org/libs/python/doc/index.html.

[2] Marc Abrams. *World Wide Web: Beyond the Basics*, chapter 1. Prentice Hall, 1998. Chapter 1, *History of the World Wide Web*, is written by Shahrooz Feizabadi.

[3] David Ascher, Paul F. Dubois, Konrad Hinsen, Jim Hugunin, and Travis Oliphant. *Numerical Python Documentation*, 2001. See http://www.pfdubois.com/numpy/html/numdoc.htm.

[4] David M. Beazley. *SWIG 1.1 User's manual*. See http://swig.sourceforge.net/Doc1.1/HTML/Contents.html.

[5] David M. Beazley. *Python Essential Reference*. New Riders, 201 West 103rd Street, Indianapolis, IN 46290, 1999.

[6] David M. Beazley and P.S. Lomdahl. Lightweight computational steering of very large scale molecular dynamics simulations. In *Supercomputing'96*. IEEE Computer Society, 1996.

[7] Neil Bradley. *The XML Companion*. Addison-Wesley, Reading, Massachusetts, 2000.

[8] Python Community. *Python Documentation*, 2000. See http://www.python.org/doc/.

[9] World Wide Web Consortium. *Extensible Markup Language (XML) 1.0. W3C Recommendation*, 2000. See http://www.w3.org/TR/2000/REC-xml-20001006.

[10] Paul F. Dubois. *Pyfort Reference Manual*. Lawrence Livermore National Laboratory. See http://pyfortran.sourceforge.net/pyfort_reference.html.

[11] Paul F. Dubois. *Writing Python Extensions in C++*. See http://cxx.sourceforge.net/.

[12] Carlo Ghezzi and Mehdi Jazayeri. *Programming Lanuage Concepts*. John Wiley & Sons, New York, 1998.

[13] J. Edward Swan II, Marco Lanzagorta, Doug Maxwell, Eddy Kuo, Jeff Uhlmann, Wendell Anderson, Haw-Jye Shyu, and William Smith. A computational steering system for studying microwave interactions with space-borne bodies. In

Proceedings IEEE Visualization 2000. IEEE Computer Society, 2000. See http://www.ait.nrl.navy.mil/vrlab/projects/CompSteering/CompSteering.html.

[14] Brian Ingerson. *Pathologically Polluting Perl*. Perl Inline module documentation. See http://www.perl.com/pub/2001/02/inline.html.

[15] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, New Jersey 07632, 1988. This is the C Bible.

[16] Hans Petter Langtangen. *Computational Partial Differential Equations*. Springer, Berlin, Heidelberg, New York, 1999.

[17] Hans Petter Langtangen. Scripting and high-performance computing. 2002.

[18] Randall J. LeVeque. *CLAWPACK Version 4.0 User's Guide*. See http://www.amath.washington.edu/ claw/.

[19] John K. Ousterhout. Scripting: Higher-level programming for the 21st century. *IEEE Computer Magazine*, March 1998. See http://www.scriptics.com/people/john.ousterhout/scripting.html.

[20] Pearu Peterson. *f2py User's Guide: Fortran to Python Interface Generator*, 2 edition, 2000. See http://cens.ioc.ee/projects/f2py2e/usersguide.html.

[21] Lutz Prechelt. An empirical comparison of c, c++, java, perl, python, rexx and tcl for a search/string-processing program. Technical Report 5, Fakultät für Informatik, Universität Karlsruhe, march 2000.

[22] Vetle Roeim and Hans Petter Langtangen. Implementation of a wave simulator using objects in c++. Technical report, 2000. This is an early draft version.

[23] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Massachusetts, 1997.

[24] Guido van Rossum and Fred L. Drake Jr. *Python 2.0 documentation: Extending and Embedding the Python Interpreter*, 2000.

Appendix A

Source code

The packages and software needed for running the source code developed during the work on this thesis is listed below. Note that the code is developed and run on a Linux system. It will probably work on most Unix versions without problems, except that some paths may have to be changed. Using the code on a Windows system will probably require some work.

A.1 System requirements

Some system requirements must be satisfied for using the code in the thesis. First of all, there must be compilers for C, C++ and Fortran available. Then Perl version 5.0 or later must be installed with the C inline module. Tcl version 8.0 or later, with the tkinter package must be installed. SWIG version 1.1p5 must be installed. Python 2.0 or later must be installed with a large number of extra modules

- Numeric module version 16.0 or later
- Scientific module version 2.0.1 or later
- PyXml version 0.6.5 or later
- Pyfort version 6.0.1 or later
- FPIG (also called f2py) version 2.298 or later
- Gnuplot version 1.4 or later
- Boost version 1.19.0 or later
- CXX version 5.0 or later
- SCXX
- Siloon version 2.3 or later
- Pdtoolkit version 1.3 and later with gcc patch which can be downloaded from the Pdtoolkit home page.
- tkinter

A.2 File listing

The source code files, setup files, and makefiles needed are listed beneath. A zipped tar-file with all the files is available.

```
src/ex/swig/example.c
src/ex/swig/example.h
src/ex/swig/example.i
src/ex/swig/clean.sh
src/ex/swig/make.sh
src/ex/swig/test.py
src/ex/fillarr/runall
src/ex/fillarr/C-F77/bench.py
src/ex/fillarr/C-F77/fillarr.c
src/ex/fillarr/C-F77/fillarr.f
src/ex/fillarr/C-F77/make.sh
src/ex/fillarr/DpC++/Makefile
src/ex/fillarr/DpC++/fillarr.cpp
src/ex/fillarr/DpC++/fillarr_f77.f
src/ex/fillarr/NumPy/fillarr1_numpy.py
src/ex/fillarr/NumPy/fillarr2_numpy.py
src/ex/fillarr/NumPy/numpy_basics.py
src/ex/fillarr/NumPy/C/basic/Makefile
src/ex/fillarr/NumPy/C/basic/fillarr_Numeric_basic_C.py
src/ex/fillarr/NumPy/C/basic/fillarr_basic_C.c
src/ex/fillarr/NumPy/C/basic/make.sh
src/ex/fillarr/NumPy/C/basic/clean.sh
src/ex/fillarr/NumPy/C/swig/Makefile
src/ex/fillarr/NumPy/C/swig/Makefile.template
src/ex/fillarr/NumPy/C/swig/fillarr_Numeric_swig_C.py
src/ex/fillarr/NumPy/C/swig/fillarr_swig_C.i
src/ex/fillarr/NumPy/C/swig/make.sh
src/ex/fillarr/NumPy/CXX/Makefile
src/ex/fillarr/NumPy/CXX/fillarr.cxx
src/ex/fillarr/NumPy/CXX/test.py
src/ex/fillarr/NumPy/F77/f2py/Makefile
src/ex/fillarr/NumPy/F77/f2py/fillarr_Numeric_f77.py
src/ex/fillarr/NumPy/F77/f2py/make.sh
src/ex/fillarr/NumPy/F77/pyfort/Makefile
src/ex/fillarr/NumPy/F77/pyfort/fillarr.pyf
src/ex/fillarr/NumPy/F77/pyfort/fillarr_Numeric_f77.py
src/ex/fillarr/NumPy/makeall
src/ex/fillarr/NumPy/cleanall
src/ex/fillarr/script/fillarr.pl
src/ex/fillarr/script/fillarr.py
src/ex/fillarr/script/fillarr.tcl
src/ex/fillarr/makeall
src/ex/fillarr/clean.sh
src/ex/makeall
src/ex/cleanall
src/wave/LongWave1D/python/Bottom.py
src/wave/LongWave1D/python/Surface.py
src/wave/LongWave1D/python/wave.py
src/wave/LongWave1D/python/CBottom2/Bottom.py
src/wave/LongWave1D/python/CBottom2/CBottommodule.c
src/wave/LongWave1D/python/CBottom2/Makefile
src/wave/LongWave1D/python/CBottom2/wave.py
src/wave/LongWave1D/python/CBottom2/gui.py
src/wave/LongWave1D/python/CBottom2/waveSteering.py
src/wave/LongWave1D/python/CSolve/Bottom.py
src/wave/LongWave1D/python/CSolve/csolvemodule.c
src/wave/LongWave1D/python/CSolve/Makefile
src/wave/LongWave1D/python/CSolve/wave2.py
src/wave/LongWave1D/python/CSolve/gui.py
src/wave/LongWave1D/python/CSolve/waveSteering.py
src/wave/LongWave1D/python/CBottom/Makefile
src/wave/LongWave1D/python/CBottom/CBottommodule.c
src/wave/LongWave1D/python/CBottom/Bottom.py
```

```
src/wave/LongWave1D/python/CBottom/wave.py
src/wave/LongWave1D/python/CBottom/waveSteering.py
src/wave/LongWave1D/python/CBottom/gui.py
src/wave/LongWave1D/python/waveSteering.py
src/wave/LongWave1D/python/gui.py
src/wave/LongWave1D/python/xml/input.dtd
src/wave/LongWave1D/python/xml/input.xml
src/wave/LongWave1D/python/xml/output.xml
src/wave/LongWave1D/python/xml/output.xsl
src/wave/LongWave1D/python/wave2.py
src/wave/LongWave1D/python/NumPyDB.py
src/wave/LongWave1D/python/Bottom2.py
src/wave/LongWave1D/F77/prog.f
src/wave/LongWave1D/F77/f2py/prog.f90
src/wave/LongWave1D/F77/f2py/Makefile
src/wave/LongWave1D/plain-C/main.c
src/wave/LongWave1D/plain-C/wave1D.c
src/wave/LongWave1D/plain-C/wave1D.h
src/wave/LongWave1D/plain-C/Makefile
src/wave/LongWave1D/plain-C/plot.py
src/wave/LongWave1D/Diffpack-C++/Makefile
src/wave/LongWave1D/Diffpack-C++/README
src/wave/LongWave1D/Diffpack-C++/Wave1D6.cpp
src/wave/LongWave1D/Diffpack-C++/Wave1D6.h
src/wave/LongWave1D/Diffpack-C++/main.cpp
src/wave/LongWave1D/Diffpack-C++/alt/Makefile
src/wave/LongWave1D/Diffpack-C++/alt/README
src/wave/LongWave1D/Diffpack-C++/alt/Wave1D6.cpp
src/wave/LongWave1D/Diffpack-C++/alt/Wave1D6.h
src/wave/LongWave1D/Diffpack-C++/alt/main.cpp
src/wave/LongWave1D/README
src/wave/Wave2D/python/Wave2D1.py
src/wave/Wave2D/python/Surface.py
src/wave/Wave2D/python/Bottom.py
src/wave/Wave2D/C++/CommandLineArgs.h
src/wave/Wave2D/C++/Convert2vtk.h
src/wave/Wave2D/C++/FieldLattice.h
src/wave/Wave2D/C++/GridLattice.h
src/wave/Wave2D/C++/Handle.h
src/wave/Wave2D/C++/MyArray.h
src/wave/Wave2D/C++/TimePrm.h
src/wave/Wave2D/C++/Visualizer.h
src/wave/Wave2D/C++/Wave2D1.h
src/wave/Wave2D/C++/WaveFunc.h
src/wave/Wave2D/C++/real.h
src/wave/Wave2D/C++/CommandLineArgs.cpp
src/wave/Wave2D/C++/Convert2vtk.cpp
src/wave/Wave2D/C++/FieldLattice.cpp
src/wave/Wave2D/C++/GridLattice.cpp
src/wave/Wave2D/C++/MyArray.cpp
src/wave/Wave2D/C++/Visualizer.cpp
src/wave/Wave2D/C++/Wave2D1.cpp
src/wave/Wave2D/C++/WaveFunc.cpp
src/wave/Wave2D/C++/handle_test.cpp
src/wave/Wave2D/C++/main.cpp
src/wave/Wave2D/C++/test.cpp
src/wave/Wave2D/C++/bpl-MyArray/Makefile
src/wave/Wave2D/C++/bpl-MyArray/testmodule.cpp
src/wave/Wave2D/C++/images/template.param
src/wave/Wave2D/C++/make.sh
src/wave/Wave2D/C++/makefiles/Makefile-Vetle
src/wave/Wave2D/C++/makefiles/Makefile-hpl
src/wave/Wave2D/C++/siloon-MyArray/MODULE
src/wave/Wave2D/C++/siloon-MyArray/Makefile
src/wave/Wave2D/C++/siloon-MyArray/user.defs
src/wave/Wave2D/C++/siloon-MyArray/prototypes.doinclude
src/wave/Wave2D/C++/siloon-MyArray/python/siloon_python.cc
src/wave/Wave2D/C++/siloon-MyArray/python/Makefile
src/wave/Wave2D/C++/siloon-MyArray/python/MyArray.py
```

```
src/wave/Wave2D/C++/siloon-MyArray/Make.dynamic
src/wave/Wave2D/C++/siloon-MyArray/MyArray_siloon.pdb
src/wave/Wave2D/C++/siloon-MyArray/prototypes.included
src/wave/Wave2D/C++/siloon-MyArray/prototypes.excluded
src/wave/Wave2D/C++/siloon-MyArray/prototypes.unsupported
src/wave/Wave2D/C++/siloon-MyArray/siloon_includes.h
src/wave/Wave2D/C++/siloon-MyArray/siloon_register.cc
src/wave/Wave2D/C++/siloon-MyArray/siloon_execute.h
src/wave/Wave2D/C++/siloon-MyArray/siloon_execute.cc
src/wave/Wave2D/C++/clean.sh
src/wave/makeall
src/wave/cleanall
src/sbeam/Statistics.py
src/sbeam/cleanall
src/sbeam/makeall
src/sbeam/plot.py
src/sbeam/runall
src/sbeam/sbeam_numpy.py
src/sbeam/sbeam_py.py
src/sbeam/tests.verify
src/sbeam/C/MC.cpp
src/sbeam/C/MC.h
src/sbeam/C/Statistics.cpp
src/sbeam/C/Statistics.h
src/sbeam/C/clean
src/sbeam/C/draw.c
src/sbeam/C/draw.h
src/sbeam/C/main.cpp
src/sbeam/C/make.sh
src/sbeam/C/tests.verify
src/sbeam/C/swig-MonteCarlo/Makefile.swig
src/sbeam/C/swig-MonteCarlo/Setup.sbeam
src/sbeam/C/swig-MonteCarlo/clean
src/sbeam/C/swig-MonteCarlo/make1.sh
src/sbeam/C/swig-MonteCarlo/make2.sh
src/sbeam/C/swig-MonteCarlo/make3.sh
src/sbeam/C/swig-MonteCarlo/make4.sh
src/sbeam/C/swig-MonteCarlo/sbeam.i
src/sbeam/C/swig-MonteCarlo/sbeam_py_MonteCarlo.py
src/sbeam/C/swig-draw/Makefile.swig
src/sbeam/C/swig-draw/Setup.draw
src/sbeam/C/swig-draw/clean
src/sbeam/C/swig-draw/draw.i
src/sbeam/C/swig-draw/make1.sh
src/sbeam/C/swig-draw/make2.sh
src/sbeam/C/swig-draw/make3.sh
src/sbeam/C/swig-draw/make4.sh
src/sbeam/C/swig-draw/sbeam_py_draw.py
src/sbeam/C/swig-draw-beam/Makefile.swig
src/sbeam/C/swig-draw-beam/Setup.draw
src/sbeam/C/swig-draw-beam/clean
src/sbeam/C/swig-draw-beam/draw.i
src/sbeam/C/swig-draw-beam/make1.sh
src/sbeam/C/swig-draw-beam/make2.sh
src/sbeam/C/swig-draw-beam/make3.sh
src/sbeam/C/swig-draw-beam/make4.sh
src/sbeam/C/swig-draw-beam/sbeam_py_draw_beam.py
src/sbeam/C/siloon-MonteCarlo/MODULE
src/sbeam/C/siloon-MonteCarlo/Makefile
src/sbeam/C/siloon-MonteCarlo/user.defs
src/sbeam/C/siloon-MonteCarlo/prototypes.doinclude
src/sbeam/C/siloon-MonteCarlo/python/siloon_python.cc
src/sbeam/C/siloon-MonteCarlo/python/Makefile
src/sbeam/C/siloon-MonteCarlo/python/MonteCarlo.py
src/sbeam/C/siloon-MonteCarlo/python/MonteCarlo.pyc
src/sbeam/C/siloon-MonteCarlo/python/sbeam_py_MonteCarlo.py
src/sbeam/C/siloon-MonteCarlo/python/sbeam.py
src/sbeam/C/siloon-MonteCarlo/python/sbeam.pyc
src/sbeam/C/bpl-MonteCarlo/sbeammodule.cpp
```

```
src/sbeam/C/bpl-MonteCarlo/Makefile
src/sbeam/C/bpl-MonteCarlo/test.py
src/sbeam/Fortran/Makefile
src/sbeam/Fortran/draw.f
src/sbeam/Fortran/mc.f
src/sbeam/Fortran/xgasdev.f
src/sbeam/Fortran/xgauss.f
src/sbeam/Fortran/xmc.f
src/sbeam/Fortran/f2py-MonteCarlo/.f2py_get_compiler_CC
src/sbeam/Fortran/f2py-MonteCarlo/.f2py_get_compiler_FC
src/sbeam/Fortran/f2py-MonteCarlo/.f2py_get_compiler_LD
src/sbeam/Fortran/f2py-MonteCarlo/Makefile-MonteCarlo
src/sbeam/Fortran/f2py-MonteCarlo/MonteCarlo.pyf
src/sbeam/Fortran/f2py-MonteCarlo/clean
src/sbeam/Fortran/f2py-MonteCarlo/make.sh
src/sbeam/Fortran/f2py-MonteCarlo/sbeam_numpy_MonteCarlo.py
src/sbeam/Fortran/f2py-draw/clean
src/sbeam/Fortran/f2py-draw/make.sh
src/sbeam/Fortran/f2py-draw/sbeam_py_draw.py
src/sbeam/Fortran/f2py-draw-beam/clean
src/sbeam/Fortran/f2py-draw-beam/make.sh
src/sbeam/Fortran/f2py-draw-beam/sbeam_py_draw_beam.py
src/sbeam/Fortran/pyfort-MonteCarlo/Makefile
src/sbeam/Fortran/pyfort-MonteCarlo/MonteCarlo.pyf
src/sbeam/Fortran/pyfort-MonteCarlo/sbeam_numpy_MonteCarlo.py
src/sbeam/Fortran/pyfort-draw/Makefile
src/sbeam/Fortran/pyfort-draw/draw.pyf
src/sbeam/Fortran/pyfort-draw/sbeam_py_draw.py
src/sbeam/Fortran/pyfort-draw-beam/Makefile
src/sbeam/Fortran/pyfort-draw-beam/draw.pyf
src/sbeam/Fortran/pyfort-draw-beam/sbeam_py_draw_beam.py
src/makeall
src/INSTALL.txt
src/cleanall
```